uWSGI Documentation

Release 2.0

uWSGI

March 14, 2014

Contents

1 Included components (updated to latest stable release) 3

2 Quickstarts 5
- 2.1 Quickstart for Python/WSGI applications . . . 5
- 2.2 Quickstart for perl/PSGI applications . . . 10
- 2.3 Quickstart for ruby/Rack applications . . . 15
- 2.4 Snippets . . . 23

3 Table of Contents 25
- 3.1 Getting uWSGI . . . 25
- 3.2 Installing uWSGI . . . 25
- 3.3 The uWSGI build system . . . 26
- 3.4 Managing the uWSGI server . . . 29
- 3.5 Supported languages and platforms . . . 32
- 3.6 Supported Platforms/Systems . . . 32
- 3.7 Web server integration . . . 33
- 3.8 Frequently Asked Questions (FAQ) . . . 34
- 3.9 Things to know (best practices and "issues") . . . 38
- 3.10 Configuring uWSGI . . . 39
- 3.11 Fallback configuration . . . 45
- 3.12 Configuration logic . . . 47
- 3.13 Configuration Options . . . 50
- 3.14 Defining new options for your instances . . . 139
- 3.15 How uWSGI parses config files . . . 141
- 3.16 uwsgi protocol magic variables . . . 143
- 3.17 The uwsgi Protocol . . . 145
- 3.18 Managing external daemons/services . . . 147
- 3.19 The Master FIFO . . . 149
- 3.20 Socket activation with inetd/xinetd . . . 151
- 3.21 Running uWSGI via Upstart . . . 151
- 3.22 SystemD . . . 153
- 3.23 Running uWSGI instances with Circus . . . 155
- 3.24 Embedding an application in uWSGI . . . 156
- 3.25 Logging . . . 158
- 3.26 Formatting uWSGI requests logs . . . 163
- 3.27 Log encoders . . . 165
- 3.28 Hooks . . . 168
- 3.29 Glossary . . . 172

3.30 uWSGI third party plugins . . . 172

4 Tutorials **175**
4.1 The uWSGI Caching Cookbook . . . 175
4.2 Setting up Django and your web server with uWSGI and nginx . . . 183
4.3 Running uWSGI on Dreamhost shared hosting . . . 191
4.4 Running python webapps on Heroku with uWSGI . . . 195
4.5 Running Ruby/Rack webapps on Heroku with uWSGI . . . 199
4.6 Reliably use FUSE filesystems for uWSGI vassals (with Linux) . . . 203
4.7 Build a dynamic proxy using RPC and internal routing . . . 206
4.8 Setting up Graphite on Ubuntu using the Metrics subsystem . . . 207

5 Articles **211**
5.1 Serializing accept(), AKA Thundering Herd, AKA the Zeeg Problem . . . 211
5.2 The Art of Graceful Reloading . . . 218
5.3 Fun with Perl, Eyetoy and RaspberryPi . . . 227

6 uWSGI Subsystems **233**
6.1 The uWSGI alarm subsystem (from 1.3) . . . 233
6.2 The uWSGI caching framework . . . 237
6.3 WebCaching framework . . . 240
6.4 The uWSGI cron-like interface . . . 242
6.5 The uWSGI FastRouter . . . 245
6.6 uWSGI internal routing . . . 248
6.7 The uWSGI Legion subsystem . . . 261
6.8 Locks . . . 265
6.9 uWSGI Mules . . . 266
6.10 The uWSGI offloading subsystem . . . 267
6.11 The uWSGI queue framework . . . 268
6.12 uWSGI RPC Stack . . . 270
6.13 SharedArea – share memory pages between uWSGI components . . . 272
6.14 The uWSGI Signal Framework . . . 274
6.15 The uWSGI Spooler . . . 277
6.16 uWSGI Subscription Server . . . 281
6.17 Serving static files with uWSGI (updated to 1.9) . . . 284
6.18 SNI - Server Name Identification (virtual hosting for SSL nodes) . . . 288
6.19 The GeoIP plugin . . . 290
6.20 uWSGI Transformations . . . 291
6.21 WebSocket supports . . . 294
6.22 The Metrics subsystem . . . 296
6.23 The Chunked input API . . . 304

7 Scaling with uWSGI **307**
7.1 The uWSGI cheaper subsystem – adaptive process spawning . . . 307
7.2 The uWSGI Emperor – multi-app deployment . . . 311
7.3 Auto-scaling with Broodlord mode . . . 321
7.4 Zerg mode . . . 321
7.5 Adding applications dynamically . . . 323
7.6 Scaling SSL connections (uWSGI 1.9) . . . 324

8 Securing uWSGI **329**
8.1 Setting POSIX Capabilities . . . 329
8.2 Running uWSGI in a Linux CGroup . . . 330
8.3 Using Linux KSM in uWSGI . . . 331
8.4 Jailing your apps using Linux Namespaces . . . 332

8.5 The old way: the –namespace option . . . 334
8.6 FreeBSD Jails . . . 336
8.7 The Forkpty Router . . . 340
8.8 The TunTap Router . . . 342

9 Keeping an eye on your apps **345**
9.1 Monitoring uWSGI with Nagios . . . 345
9.2 The embedded SNMP server . . . 345
9.3 Pushing statistics (from 1.4) . . . 346
9.4 Integration with Graphite/Carbon . . . 347
9.5 The uWSGI Stats Server . . . 348
9.6 The Metrics subsystem . . . 352

10 Async and loop engines **361**
10.1 uWSGI asynchronous/non-blocking modes (updated to uWSGI 1.9) . . . 361
10.2 The Gevent loop engine . . . 364
10.3 The Tornado loop engine . . . 366
10.4 uGreen – uWSGI Green Threads . . . 369

11 Web Server support **371**
11.1 Apache support . . . 371
11.2 Cherokee support . . . 372
11.3 Native HTTP support . . . 373
11.4 HTTPS support (from 1.3) . . . 376
11.5 The SPDY router (uWSGI 1.9) . . . 377
11.6 Lighttpd support . . . 378
11.7 Attaching uWSGI to Mongrel2 . . . 379
11.8 Nginx support . . . 381

12 Language support **385**
12.1 Python support . . . 385
12.2 The PyPy plugin . . . 404
12.3 Running PHP scripts in uWSGI . . . 413
12.4 uWSGI Perl support (PSGI) . . . 417
12.5 Ruby support . . . 420
12.6 Using Lua/WSAPI with uWSGI . . . 427
12.7 JVM in the uWSGI server (updated to 1.9) . . . 431
12.8 The Mono ASP.NET plugin . . . 441
12.9 Running CGI scripts on uWSGI . . . 443
12.10 The gccgo plugin . . . 447
12.11 The Symcall plugin . . . 450
12.12 The XSLT plugin . . . 453
12.13 SSI (Server Side Includes) plugin . . . 454
12.14 uWSGI V8 support . . . 456
12.15 The GridFS plugin . . . 457
12.16 The GlusterFS plugin . . . 461
12.17 The RADOS plugin . . . 464

13 Other plugins **467**
13.1 The Pty plugin . . . 467
13.2 SPNEGO authentication . . . 468
13.3 Configuring uWSGI with LDAP . . . 468

14 Broken/deprecated features **469**
14.1 Integrating uWSGI with Erlang . . . 469

14.2 Management Flags 472
14.3 uWSGI Go support (1.4 only) 473

15 Release Notes **479**
15.1 Stable releases 479
15.2 LTS releases 550

16 Contact **553**

17 Commercial support **555**

18 Donate **557**

19 Indices and tables **559**

Python Module Index **561**

The uWSGI project aims at developing a full stack for building hosting services.

Application servers (for various programming languages and protocols), proxies, process managers and monitors are all implemented using a common api and a common configuration style.

Thanks to its pluggable architecture it can be extended to support more platforms and languages.

Currently, you can write plugins in C, C++ and Objective-C.

The "WSGI" part in the name is a tribute to the namesake Python standard, as it has been the first developed plugin for the project.

Versatility, performance, low-resource usage and reliability are the strengths of the project (and the only rules followed).

CHAPTER 1

Included components (updated to latest stable release)

The Core (implements configuration, processes management, sockets creation, monitoring, logging, shared memory areas, ipc, cluster membership and the *uWSGI Subscription Server*)

Request plugins (implement application server interfaces for various languages and platforms: WSGI, PSGI, Rack, Lua WSAPI, CGI, PHP, Go ...)

Gateways (implement load balancers, proxies and routers)

The *Emperor* (implements massive instances management and monitoring)

Loop engines (implement events and concurrency, components can be run in preforking, threaded, asynchronous/evented and green thread/coroutine modes. Various technologies are supported, including uGreen, Greenlet, Stackless, *Gevent*, Coro::AnyEvent, *Tornado*, Goroutines and Fibers)

Note: uWSGI is a very active project with a fast release cycle. For this reason the code and the documentation may not always be in sync. We try to make our best to have good documentation but it is an hard work. Sorry for that. If you are in trouble, the mailing list is the best source for help regarding uWSGI. Contributors for documentation (in addition to code) are always welcome.

CHAPTER 2

Quickstarts

2.1 Quickstart for Python/WSGI applications

This quickstart will show you how to deploy simple WSGI applications and common web frameworks.

Python here is meant as CPython, for PyPy you need to use the specific plugin: *The PyPy plugin*, Jython support is on work.

Note: You need at least uWSGI 1.4 to follow the quickstart. Anything older is no more maintained and is highly buggy!

2.1.1 Installing uWSGI with Python support

Tip: When you start learning uWSGI, try to build from official sources: using distro-supplied packages may bring you a lot of headaches. When things are clear, you can use modular builds (like the ones available in your distro).

uWSGI is a (big) C application, so you need a C compiler (like the gcc or clang) and Python development headers.

On a Debian-based distro an

```
apt-get install build-essential python-dev
```

will be enough.

You have various ways to install uWSGI for Python:

- via pip

  ```
  pip install uwsgi
  ```

- using the network installer

  ```
  curl http://uwsgi.it/install | bash -s default /tmp/uwsgi
  ```

  (this will install the uWSGI binary into `/tmp/uwsgi`, feel free to change it).

- via downloading a source tarball and "making" it

  ```
  wget http://projects.unbit.it/downloads/uwsgi-latest.tar.gz
  tar zxvf uwsgi-latest.tar.gz
  cd <dir>
  make
  ```

(after the build you will have a `uwsgi` binary in the current directory).

Installing via your package distribution is not covered (would be impossibile to make everyone happy), but all of the general rules apply.

One thing you may want to take into account when testing this quickstart with distro-supplied packages, is that very probably your distribution has built uWSGI in modular way (every feature is a different plugin that must be loaded). To complete this quickstart, you have to prepend `--plugin python,http` to the first series of examples, and `--plugin python` when the HTTP router is removed (it could make no sense for you, just continue reading).

2.1.2 The first WSGI application

Let's start with a simple "Hello World" example (this is for Python 2.x, Python 3.x requires the returned string to be bytes, see lower):

```
def application(env, start_response):
    start_response('200 OK', [('Content-Type','text/html')])
    return ["Hello World"]
```

(save it as `foobar.py`).

As you can see, it is composed of a single Python function. It is called "application" as this is default function that the uWSGI Python loader will search for (but you can obviously customize it).

The Python 3.x version is the following:

```
def application(env, start_response):
    start_response('200 OK', [('Content-Type','text/html')])
    return [b"Hello World"]
```

2.1.3 Deploy it on HTTP port 9090

Now start uWSGI to run an HTTP server/router passing requests to your WSGI application:

```
uwsgi --http :9090 --wsgi-file foobar.py
```

That's all.

2.1.4 Adding concurrency and monitoring

The first tuning you would like to make is adding concurrency (by default uWSGI starts with a single process and a single thread).

You can add more processes with the `--processes` option or more threads with the `--threads` options (or you can have both).

```
uwsgi --http :9090 --wsgi-file foobar.py --master --processes 4 --threads 2
```

This will spawn 4 processes (each with 2 threads), a master process (will respawn your processes when they die) and the HTTP router (seen before).

One important task is monitoring. Understanding what is going on is vital in production deployment. The stats subsystem allows you to export uWSGI's internal statistics as JSON:

```
uwsgi --http :9090 --wsgi-file foobar.py --master --processes 4 --threads 2 --stats 127.0.0.1:9191
```

Make some request to your app and then telnet to the port 9191, you'll get a lot of funny information. You may want to use "uwsgitop" (just `pip install` it), which is a top-like tool for monitoring instances.

Attention: Bind the stats socket to a private address (unless you know what you are doing), otherwise everyone could access it!

2.1.5 Putting behind a full webserver

Even though uWSGI HTTP router is solid and high-performance, you may want to put your application behind a fully-capable webserver.

uWSGI natively speaks HTTP, FastCGI, SCGI and its specific protocol named "uwsgi" (yes, wrong naming choice). The best performing protocol is obviously uwsgi, already supported by nginx and Cherokee (while various Apache modules are available).

A common nginx config is the following:

```
location / {
    include uwsgi_params;
    uwsgi_pass 127.0.0.1:3031;
}
```

This means "pass every request to the server bound to port 3031 speaking the uwsgi protocol".

Now we can spawn uWSGI to natively speak the uwsgi protocol:

```
uwsgi --socket 127.0.0.1:3031 --wsgi-file foobar.py --master --processes 4 --threads 2 --stats 127.0.
```

If you'll run `ps aux`, you will see one process less. The HTTP router has been removed as our "workers" (the processes assigned to uWSGI) natively speak the uwsgi protocol.

2.1.6 Automatically starting uWSGI on boot

If you think about writing some init.d script for spawning uWSGI, just sit (and calm) down and check if your system offers you a better (more modern) approach.

Each distribution has chosen its startup system (*Upstart*, *SystemD*, etc.) and there are tons of process managers available: Supervisor, god, Circus, etc.

uWSGI will integrate very well with all of them (we hope), but if you plan to deploy a big number of apps check the uWSGI *Emperor*, it is the dream of every devops.

2.1.7 Deploying Django

Django is very probably the most used Python web framework around. Deploying it is pretty easy (we continue our configuration with 4 processes with 2 threads each).

We suppose the Django project is in `/home/foobar/myproject`:

```
uwsgi --socket 127.0.0.1:3031 --chdir /home/foobar/myproject/ --wsgi-file myproject/wsgi.py --master
```

(with `--chdir` we move to a specific directory). In Django this is required to correctly load modules.

If the file `/home/foobar/myproject/myproject/wsgi.py` (or whatever you have called your project) does not exist, you are very probably using an old (< 1.4) version of Django. In such a case you need a little bit more configuration:

```
uwsgi --socket 127.0.0.1:3031 --chdir /home/foobar/myproject/ --pythonpath .. --env DJANGO_SETTINGS_M
```

ARGH! What the hell is this?! Yes, you are right, dealing with such long command lines is basically unpractical (and foolish). uWSGI supports various configuration styles. In this quickstart we will use .ini files.

```
[uwsgi]
socket = 127.0.0.1:3031
chdir = /home/foobar/myproject/
pythonpath = ..
env = DJANGO_SETTINGS_MODULE=myproject.settings
module = django.core.handlers.wsgi:WSGIHandler()
processes = 4
threads = 2
stats = 127.0.0.1:9191
```

A lot better!

Just run it:

```
uwsgi yourfile.ini
```

Older (< 1.4) Django releases need to set `env`, `module` and the `pythonpath` (`..` allow us to reach the `myproject.settings` module).

2.1.8 Deploying Flask

Flask is popular Python web microframework.

Save the following example as `myflaskapp.py`:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "<span style='color:red'>I am app 1</span>"
```

Flask exports its WSGI function (the one we called "application" at the beginning of this quickstart) as "app", so we need to instruct uWSGI to use it. We still continue to use the 4 processes/2 threads and the uwsgi socket as the base:

```
uwsgi --socket 127.0.0.1:3031 --wsgi-file myflaskapp.py --callable app --processes 4 --threads 2 --st
```

(the only addition is the `--callable` option).

2.1.9 Deploying web2py

Again a popular choice. Unzip the web2py source distribution on a directory of choice and write a uWSGI config file:

```
[uwsgi]
http = :9090
chdir = path_to_web2py
module = wsgihandler
master = true
processes = 8
```

Note: On recent web2py releases you may need to copy the `wsgihandler.py` script out of the `handlers`

directory.

We used the HTTP router again. Just go to port 9090 with your browser and you will see the web2py welcome page.

Click on the administrative interface and... oops, it does not work as it requires HTTPS. Do not worry, the uWSGI router is HTTPS-capable (be sure you have OpenSSL development headers: install them and rebuild uWSGI, the build system will automatically detect it).

First of all generate your key and certificate:

```
openssl genrsa -out foobar.key 2048
openssl req -new -key foobar.key -out foobar.csr
openssl x509 -req -days 365 -in foobar.csr -signkey foobar.key -out foobar.crt
```

Now you have 2 files (well 3, counting the `foobar.csr`), `foobar.key` and `foobar.crt`. Change the uWSGI config:

```
[uwsgi]
https = :9090,foobar.crt,foobar.key
chdir = path_to_web2py
module = wsgihandler
master = true
processes = 8
```

Re-run uWSGI and connect to port 9090 using `https://` with your browser.

2.1.10 A note on Python threads

If you start uWSGI without threads, the Python GIL will not be enabled, so threads generated by your application will never run. You may not like that choice, but remember that uWSGI is a language-independent server, so most of its choices are for maintaining it "agnostic".

But do not worry, there are basically no choices made by the uWSGI developers that cannot be changed with an option.

If you want to maintain Python threads support without starting multiple threads for your application, just add the `--enable-threads` option (or `enable-threads = true` in ini style).

2.1.11 Virtualenvs

uWSGI can be configured to search for Python modules in a specific virtualenv.

Just add `virtualenv = <path>` to your options.

2.1.12 Security and availability

Always avoid running your uWSGI instances as root. You can drop privileges using the `uid` and `gid` options:

```
[uwsgi]
https = :9090,foobar.crt,foobar.key
uid = foo
gid = bar
chdir = path_to_web2py
module = wsgihandler
master = true
processes = 8
```

If you need to bind to privileged ports (like 443 for HTTPS), use shared sockets. They are created before dropping privileges and can be referenced with the `=N` syntax, where `N` is the socket number (starting from 0):

```
[uwsgi]
shared-socket = :443
https = =0,foobar.crt,foobar.key
uid = foo
gid = bar
chdir = path_to_web2py
module = wsgihandler
master = true
processes = 8
```

A common problem with webapp deployment is "stuck requests". All of your threads/workers are stuck (blocked on request) and your app cannot accept more requests. To avoid that problem you can set a `harakiri` timer. It is a monitor (managed by the master process) that will destroy processes stuck for more than the specified number of seconds (choose `harakiri` value carefully). For example, you may want to destroy workers blocked for more than 30 seconds:

```
[uwsgi]
shared-socket = :443
https = =0,foobar.crt,foobar.key
uid = foo
gid = bar
chdir = path_to_web2py
module = wsgihandler
master = true
processes = 8
harakiri = 30
```

In addition to this, since uWSGI 1.9, the stats server exports the whole set of request variables, so you can see (in realtime) what your instance is doing (for each worker, thread or async core).

2.1.13 Offloading

The uWSGI offloading subsystem allows you to free your workers as soon as possible when some specific pattern matches and can be delegated to a pure-C thread (sending static file from the filesystem, transferring data from the network to the client and so on).

Offloading is very complex, but its use is transparent to the end user. If you want to try, just add `--offload-threads <n>`, where `<n>` is the number of threads to spawn (one per CPU is a good value).

When offload threads are enabled, all of the parts that can be optimized will be automatically detected.

2.1.14 And now

You should already be able to go in production with such few concepts, but uWSGI is an enormous project with hundreds of features and configurations. If you want to be a better sysadmin, continue reading the full docs.

2.2 Quickstart for perl/PSGI applications

The following instructions will guide you through installing and running a perl-based uWSGI distribution, aimed at running PSGI apps.

2.2.1 Installing uWSGI with Perl support

To build uWSGI you need a c compiler (gcc and clang are supported) and the python binary (it will only run the uwsgiconfig.py script that will execute the various compilation steps). As we are building a uWSGI binary with perl support we need perl development headers too (libperl-dev package on debian-based distros)

You can build uWSGI manually:

```
python uwsgiconfig.py --build psgi
```

that is the same as

```
UWSGI_PROFILE=psgi make
```

or using the network installer:

```
curl http://uwsgi.it/install | bash -s psgi /tmp/uwsgi
```

that will create a uWSGI binary in /tmp/uwsgi (feel free to change the path to whatever you want)

2.2.2 Note for distro packages

You distribution very probably contains a uWSGI package set. Those uWSGI packages tend to be highly modulars, so in addition to the core you need to install the required plugins. Plugins must be loaded in your configs. In the learning phase we strongly suggest to not use distribution packages to easily follow documentation and tutorials.

Once you feel confortable with the "uWSGI way" you can choose the best approach for your deployments.

2.2.3 Your first PSGI app

save it to a file named myapp.pl

```
my $app = sub {
     my $env = shift;
     return [
             '200',
             [ 'Content-Type' => 'text/html' ],
             [ "<h1>Hello World</h1>" ],
     ];
};
```

then run it via uWSGI in http mode:

```
uwsgi --http :8080 --http-modifier1 5 --psgi myapp.pl
```

(remember to replace 'uwsgi' if it is not in your current $PATH)

or if you are using a modular build (like the one of your distro)

```
uwsgi --plugins http,psgi --http :8080 --http-modifier1 5 --psgi myapp.pl
```

2.2.4 What is that '–http-modifier1 5' thing ???

uWSGI supports various languages and platform. When the server receives a request it has to know where to 'route' it.

Each uWSGI plugin has an assigned number (the modifier), the perl/psgi one has the 5. So –http-modifier1 5 means "route to the psgi plugin"

Albeit uWSGI has a more "human-friendly" *internal routing system* using modifiers is the fastest way, so, if possible always use them

2.2.5 Using a full webserver: nginx

The supplied http router, is (yes, incredible) only a router. You can use it as a load balancer or a proxy, but if you need a full webserver (for efficiently serving static files or all of those task a webserver is good at), you can get rid of the uwsgi http router (remember to change –plugins http,psgi to –plugins psgi if you are using a modular build) and put your app behind nginx.

To communicate with nginx, uWSGI can use various protocol: http, uwsgi, fastcgi, scgi...

The most efficient one is the uwsgi one. Nginx includes uwsgi protocol support out of the box.

Run your psgi application on a uwsgi socket:

```
uwsgi --socket 127.0.0.1:3031 --psgi myapp.pl
```

then add a location stanza in your nginx config

```
location / {
    include uwsgi_params;
    uwsgi_pass 127.0.0.1:3031;
    uwsgi_modifier1 5;
}
```

Reload your nginx server, and it should start proxying requests to your uWSGI instance

Note that you do not need to configure uWSGI to set a specific modifier, nginx will do it using the `uwsgi_modifier1 5;` directive

2.2.6 Adding concurrency

You can give concurrency to to your app via multiprocess,multithreading or various async modes.

To spawn additional processes use the –processes option

```
uwsgi --socket 127.0.0.1:3031 --psgi myapp.pl --processes 4
```

To have additional threads use –threads

```
uwsgi --socket 127.0.0.1:3031 --psgi myapp.pl --threads 8
```

Or both if you feel exotic

A very common non-blocking/coroutine library in the perl world is Coro::AnyEvent. uWSGI can use it (even combined with multiprocessing) simply including the `coroae` plugin.

To build a uWSGI binary with `coroae` support just run

```
UWSGI_PROFILE=coroae make
```

or

```
curl http://uwsgi.it/install | bash -s coroae /tmp/uwsgi
```

you will end with a uWSGI binary including both the `psgi` and `coroae` plugins.

Now run your application in Coro::AnyEvent mode:

```
uwsgi --socket 127.0.0.1:3031 --psgi myapp.pl --coroae 1000 --processes 4
```

it will run 4 processes each able to manage up to 1000 coroutines (or Coro microthreads).

2.2.7 Adding robustness: the Master process

It is highly recommended to have the master process always running on productions apps.

It will constantly monitor your processes/threads and will add funny features like the *The uWSGI Stats Server*

To enable the master simply add –master

```
uwsgi --socket 127.0.0.1:3031 --psgi myapp.pl --processes 4 --master
```

2.2.8 Using config files

uWSGI has literally hundreds of options. Dealing with them via command line is basically silly, so try to always use config files. uWSGI supports various standards (xml, .ini, json, yaml...). Moving from one to another is pretty simple. The same options you can use via command line can be used on config files simply removing the `--` prefix:

```
[uwsgi]
socket = 127.0.0.1:3031
psgi = myapp.pl
processes = 4
master = true
```

or xml:

```
<uwsgi>
  <socket>127.0.0.1:3031</socket>
  <psgi>myapp.pl</psgi>
  <processes>4</processes>
  <master/>
</uwsgi>
```

To run uWSGI using a config file, just specify it as argument:

```
uwsgi yourconfig.ini
```

if for some reason your config cannot end with the expected extension (.ini, .xml, .yml, .js) you can force the binary to use a specific parser in this way:

```
uwsgi --ini yourconfig.foo

uwsgi --xml yourconfig.foo

uwsgi --yaml yourconfig.foo
```

and so on

You can even pipe configs (using the dash to force reading from stdin):

```
perl myjsonconfig_generator.pl | uwsgi --json -
```

2.2.9 Automatically starting uWSGI on boot

If you are thinking about writing some init.d script for spawning uWSGI, just sit (and calm) down and check if your system does not offer you a better (more modern) approach.

Each distribution has choosen its startup system (*Upstart*, *SystemD*...) and there are tons of process managers available (supervisord, god...).

uWSGI will integrate very well with all of them (we hope), but if you plan to deploy a big number of apps check the uWSGI *Emperor* it is the dream of every devops.

2.2.10 Security and availability

ALWAYS avoid running your uWSGI instances as root. You can drop privileges using the uid and gid options

```
[uwsgi]
socket = 127.0.0.1:3031
uid = foo
gid = bar
chdir = path_toyour_app
psgi = myapp.pl
master = true
processes = 8
```

A common problem with webapp deployment is "stuck requests". All of your threads/workers are stuck blocked on a request and your app cannot accept more of them.

To avoid that problem you can set an `harakiri` timer. It is a monitor (managed by the master process) that will destroy processes stuck for more than the specified number of seconds

```
[uwsgi]
socket = 127.0.0.1:3031
uid = foo
gid = bar
chdir = path_toyour_app
psgi = myapp.pl
master = true
processes = 8
harakiri = 30
```

will destroy workers blocked for more than 30 seconds. Choose carefully the harakiri value !!!

In addition to this, since uWSGI 1.9, the stats server exports the whole set of request variables, so you can see (in realtime) what your instance is doing (for each worker, thread or async core)

Enabling the stats server is easy:

```
[uwsgi]
socket = 127.0.0.1:3031
uid = foo
gid = bar
chdir = path_toyour_app
psgi = myapp.pl
master = true
processes = 8
harakiri = 30
stats = 127.0.0.1:5000
```

just bind it to an address (UNIX or TCP) and just connect (you can use telnet too) to it to receive a JSON representation of your instance.

The `uwsgitop` application (you can find it in the official github repository) is an example of using the stats server to have a top-like realtime monitoring tool (with colors !!!)

2.2.11 Offloading

The uWSGI offloading subsystem allows you to free your workers as soon as possible when some specific pattern matches and can be delegated to a pure-c thread. Examples are sending static file from the filesystem, transferring data from the network to the client and so on.

Offloading is very complex, but its use is transparent to the end user. If you want to try just add –offload-threads <n> where <n> is the number of threads to spawn (one for cpu is a good value).

When offload threads are enabled, all of the parts that can be optimized will be automatically detected

2.2.12 And now

You should already be able to go in production with such few concepts, but uWSGI is an enormous project with hundreds of features and configurations. If you want to be a better sysadmin, continue reading the full docs.

2.3 Quickstart for ruby/Rack applications

The following instructions will guide you through installing and running a ruby-based uWSGI distribution, aimed at running Rack apps.

2.3.1 Installing uWSGI with Ruby support

To build uWSGI you need a c compiler (gcc and clang are supported) and the python binary (it will only run the uwsgiconfig.py script that will execute the various compilation steps). As we are building a uWSGI binary with ruby support we need ruby development headers too (ruby-dev package on debian-based distros)

You can build uWSGI manually:

```
make rack
```

that is the same as

```
UWSGI_PROFILE=rack make
```

that is the same of

```
make PROFILE=rack
```

and

```
python uwsgiconfig.py --build rack
```

If you are lazy, you can download, build and install a uWSGI+ruby binary in a single shot:

```
curl http://uwsgi.it/install | bash -s rack /tmp/uwsgi
```

Or, more ruby-friendly:

```
gem install uwsgi
```

All of this methods build a "monolithic" uWSGI binary. The uWSGI project is composed by dozens of plugins, you can choose to build the server core and having a plugin for every feature (that you will load when needed), or you can build a single binary with the features you need. This kind of builds are called 'monolithic'.

This quickstart assumes a monolithic binary (so you do not need to load plugins). If you prefer to use your package distributions (instead of building uWSGI from official sources), see below

2.3.2 Note for distro packages

You distribution very probably contains a uWSGI package set. Those uWSGI packages tend to be highly modulars, so in addition to the core you need to install the required plugins. Plugins must be loaded in your configs. In the learning phase we strongly suggest to not use distribution packages to easily follow documentation and tutorials.

Once you feel confortable with the "uWSGI way" you can choose the best approach for your deployments.

As an example, the tutorial makes use of the 'http' and 'rack' plugins. If you are using a modular build be sure to load them with the `--plugins http,rack` option

2.3.3 Your first Rack app

Rack is the standard way for writing ruby web apps.

This is a standard Rack Hello world script (call it app.ru):

```
class App

  def call(environ)
    [200, {'Content-Type' => 'text/html'}, ['Hello']]
  end

end

run App.new
```

The .ru extension stands for "rackup" that is the deployment tool included in the rack distribution. Rackup uses a little DSL, so to use it into uWSGI you need to install the rack gem:

```
gem install rack
```

Now we are ready to deploy with uWSGI:

```
uwsgi --http :8080 --http-modifier1 7 --rack app.ru
```

(remember to replace 'uwsgi' if it is not in your current $PATH)

or if you are using a modular build (like the one of your distro)

```
uwsgi --plugins http,rack --http :8080 --http-modifier1 7 --rack app.ru
```

Whit this command line we spawned an http proxy routing each request to a process (named the 'worker') that manages it and send back the response to the http router (that sends back to the client).

If you are asking yourself why spawning two processes, it is because this is the normal architecture you will use in production (a frontline webserver with a backend application server).

If you do not want to spawn the http proxy and directly force the worker to answer http requests just change the command line to

```
uwsgi --http-socket :8080 --http-socket-modifier1 7 --rack app.ru
```

now you have a single process managing requests (but remember that directly exposing the application server to the public is generally dangerous and less versatile)

2.3.4 What is that '–http-modifier1 7' thing ???

uWSGI supports various languages and platforms. When the server receives a request it has to know where to 'route' it.

Each uWSGI plugin has an assigned number (the modifier), the ruby/rack one has the 7. So –http-modifier1 7 means "route to the rack plugin"

Albeit uWSGI has a more "human-friendly" *internal routing system* using modifiers is the fastest way, so, if possible always use them

2.3.5 Using a full webserver: nginx

The supplied http router, is (yes, incredible) only a router. You can use it as a load balancer or a proxy, but if you need a full webserver (for efficiently serving static files or all of those task a webserver is good at), you can get rid of the uwsgi http router (remember to change –plugins http,rack to –plugins rack if you are using a modular build) and put your app behind nginx.

To communicate with nginx, uWSGI can use various protocol: http, uwsgi, fastcgi, scgi...

The most efficient one is the uwsgi one. Nginx includes uwsgi protocol support out of the box.

Run your rack application on a uwsgi socket:

```
uwsgi --socket 127.0.0.1:3031 --rack app.ru
```

then add a location stanza in your nginx config

```
location / {
    include uwsgi_params;
    uwsgi_pass 127.0.0.1:3031;
    uwsgi_modifier1 7;
}
```

Reload your nginx server, and it should start proxying requests to your uWSGI instance

Note that you do not need to configure uWSGI to set a specific modifier, nginx will do it using the `uwsgi_modifier1 5;` directive

2.3.6 Adding concurrency

With the previous example you deployed a stack being able to serve a single request at time.

To increase concurrency you need to add more processes. If you hope there is a magic math formula to find the right number of processes to spawn, lose it. You need to experiment and monitor your app to find the right value. Take in account every single process is a complete copy of your app, so memory should be taken in account.

To add more processes just use the *–processes <n>* option:

```
uwsgi --socket 127.0.0.1:3031 --rack app.ru --processes 8
```

will spawn 8 processes.

Ruby 1.9/2.0 introduced an improved threads support and uWSGI supports it via the 'rbthreads' plugin. This plugin is automatically build when you compile the uWSGI+ruby (>=1.9) monolithic binary.

To add more threads:

```
uwsgi --socket 127.0.0.1:3031 --rack app.ru --rbthreads 4
```

or threads + processes

```
uwsgi --socket 127.0.0.1:3031 --rack app.ru --processes --rbthreads 4
```

There are other (generally more advanced/complex) ways to increase concurrency (for example 'fibers'), but most of the time you will end with plain old multiprocesses or multithreads models. (if you are interested you can check to uWSGI rack full docs)

2.3.7 Adding robustness: the Master process

It is highly recommended to have the master process always running on productions apps.

It will constantly monitor your processes/threads and will add funny features like the *The uWSGI Stats Server*

To enable the master simply add –master

```
uwsgi --socket 127.0.0.1:3031 --rack app.ru --processes 4 --master
```

2.3.8 Using config files

uWSGI has literally hundreds of options (but generally you will not use more than a dozens of them). Dealing with them via command line is basically silly, so try to always use config files. uWSGI supports various standards (xml, .ini, json, yaml...). Moving from one to another is pretty simple. The same options you can use via command line can be used on config files simply removing the −− prefix:

```
[uwsgi]
socket = 127.0.0.1:3031
rack = app.ru
processes = 4
master = true
```

or xml:

```
<uwsgi>
  <socket>127.0.0.1:3031</socket>
  <rack>app.ru</rack>
  <processes>4</processes>
  <master/>
</uwsgi>
```

To run uWSGI using a config file, just specify it as argument:

```
uwsgi yourconfig.ini
```

if for some reason your config cannot end with the expected extension (.ini, .xml, .yml, .js) you can force the binary to use a specific parser in this way:

```
uwsgi --ini yourconfig.foo
```

```
uwsgi --xml yourconfig.foo

uwsgi --yaml yourconfig.foo
```

and so on

You can even pipe configs (using the dash to force reading from stdin):

```
ruby myjsonconfig_generator.rb | uwsgi --json -
```

2.3.9 The fork() problem when you spawn multiple processes

uWSGI is "perlish", there is nothing we can do to hide this thing. Most of its choices (starting from "There's more than one way to do it") cames from the perl world (and more generally from the UNIX sysadmins approaches).

Sometimes this approach could lead to unexpected behaviours when applied to other languages/platform.

One of the "problems" you can face when starting to learn uWSGI is its fork() usage.

By default uWSGI loads your application in the first spawned process and then fork() itself multiple times.

It means your app is loaded a single time and then copied.

While this approach speedups the start of the server, some application could have problems with this technique (expecially those initializing db connections on startup, as the file descirptor of the connection will be inherited in the subprocesses)

If you are unsure about the brutal preforking used by uWSGI, just disable it with the `--lazy-apps` option. It will force uWSGI to completely load your app one time per-worker

2.3.10 Deploying Sinatra

Let's forget about fork(), and back to funny things. This time we deploy a Sinatra application:

```
require 'sinatra'

get '/hi' do
  "Hello World"
end

run Sinatra::Application
```

save it as config.ru and run as seen before:

```
[uwsgi]
socket = 127.0.0.1:3031
rack = config.ru
master = true
processes = 4
lazy-apps = true

uwsgi yourconf.ini
```

well maybe you have already noted that basically nothing changed from the previous app.ru examples.

That is because basically every modern Rack app exposes itself as a .ru file (generally called config.ru), so there is no need for multiple options for loading application (like for example in the python/WSGI world)

2.3.11 Deploying RubyOnRails >= 3

Starting from 3.0, Rails is fully rack compliant, and exposes a config.ru file you can directly load (like we did with sinatra)

The only difference from sinatra is that your project has a specific layout/convention expecting your current working directory is the one containing the project, so le'ts add a chdir option:

```
[uwsgi]
socket = 127.0.0.1:3031
rack = config.ru
master = true
processes = 4
lazy-apps = true
chdir = <path_to_your_rails_app>
env = RAILS_ENV=production

uwsgi yourconf.ini
```

in addition to chdir we have added the 'env' option that set the RAILS_ENV environment variable.

Starting from 4.0, Rails support multiple threads (only for ruby 2.0):

```
[uwsgi]
socket = 127.0.0.1:3031
rack = config.ru
master = true
processes = 4
rbthreads = 2
lazy-apps = true
chdir = <path_to_your_rails_app>
env = RAILS_ENV=production
```

2.3.12 Deploying older RubyOnRails

Older Rails versions are not fully Rack-compliant. For such a reason a specific option is available in uWSGI to load older rails app (you will need the 'thin' gem too).

```
[uwsgi]
socket = 127.0.0.1:3031
master = true
processes = 4
lazy-apps = true
rails = <path_to_your_rails_app>
env = RAILS_ENV=production
```

the 'rails' options must be specified instead of 'rack' passing the rails app directory as the argument

2.3.13 Bundler and RVM

Bundler is the standard-de-facto ruby tool for managing dependancies. Basically you specify the gem needed by your app in the Gemfile text file and then you launch bundler to install them.

To allow uWSGI to honour bundler installations you only need to add:

```
rbrequire = rubygems
rbrequire = bundler/setup
env = BUNDLE_GEMFILE=<path_to_your_Gemfile>
```

the first line is not required for ruby 1.9/2.x

Basically those lines force uWSGI to load the bundler engine and to use the Gemfile specified in the BUNDLE_GEMFILE environment variable.

When using Bundler (like modern frameworks do) your common deployment configuration will be:

```
[uwsgi]
socket = 127.0.0.1:3031
rack = config.ru
master = true
processes = 4
lazy-apps = true
rbrequire = rubygems
rbrequire = bundler/setup
env = BUNDLE_GEMFILE=<path_to_your_Gemfile>
```

In addition to Bundler, RVM is another common tool.

It allows you to have multiple (independent) ruby installations (with their gemsets) on a single system.

To instruct uWSGI to use the gemset of a specific rvm version just use the *–gemset* option:

```
[uwsgi]
socket = 127.0.0.1:3031
rack = config.ru
master = true
processes = 4
lazy-apps = true
rbrequire = rubygems
rbrequire = bundler/setup
env = BUNDLE_GEMFILE=<path_to_your_Gemfile>
gemset = ruby-2.0@foobar
```

just pay attention you need a uWSGI binary (or a plugin if you are using a modular build) for every ruby version (ruby version, not gemset !!!)

If you are interested this is a list of commands to build a uWSGI core + 1 one plugin for every ruby version installed in rvm:

```
# build the core
make nolang
# build plugin for 1.8.7
rvm use 1.8.7
./uwsgi --build-plugin "plugins/rack rack187"
# build for 1.9.2
rvm use 1.9.2
./uwsgi --build-plugin "plugins/rack rack192"
# and so on...
```

then if you want to use ruby 1.9.2 with the @oops gemset:

```
[uwsgi]
plugins = ruby192
socket = 127.0.0.1:3031
rack = config.ru
master = true
```

```
processes = 4
lazy-apps = true
rbrequire = rubygems
rbrequire = bundler/setup
env = BUNDLE_GEMFILE=<path_to_your_Gemfile>
gemset = ruby-1.9.2@oops
```

2.3.14 Automatically starting uWSGI on boot

If you are thinking about writing some init.d script for spawning uWSGI, just sit (and calm) down and check if your system does not offer you a better (more modern) approach.

Each distribution has choosen its startup system (*Upstart*, *SystemD*...) and there are tons of process managers available (supervisord, god...).

uWSGI will integrate very well with all of them (we hope), but if you plan to deploy a big number of apps check the uWSGI *Emperor* it is the dream of every devops.

2.3.15 Security and availability

ALWAYS avoid running your uWSGI instances as root. You can drop privileges using the uid and gid options

```
[uwsgi]
socket = 127.0.0.1:3031
uid = foo
gid = bar
chdir = path_toyour_app
rack = app.ru
master = true
processes = 8
```

A common problem with webapp deployment is "stuck requests". All of your threads/workers are stuck blocked on a request and your app cannot accept more of them.

To avoid that problem you can set an `harakiri` timer. It is a monitor (managed by the master process) that will destroy processes stuck for more than the specified number of seconds

```
[uwsgi]
socket = 127.0.0.1:3031
uid = foo
gid = bar
chdir = path_toyour_app
rack = app.ru
master = true
processes = 8
harakiri = 30
```

will destroy workers blocked for more than 30 seconds. Choose carefully the harakiri value !!!

In addition to this, since uWSGI 1.9, the stats server exports the whole set of request variables, so you can see (in realtime) what your instance is doing (for each worker, thread or async core)

Enabling the stats server is easy:

```
[uwsgi]
socket = 127.0.0.1:3031
uid = foo
```

```
gid = bar
chdir = path_toyour_app
rack = app.ru
master = true
processes = 8
harakiri = 30
stats = 127.0.0.1:5000
```

just bind it to an address (UNIX or TCP) and just connect (you can use telnet too) to it to receive a JSON representation of your instance.

The `uwsgitop` application (you can find it in the official github repository) is an example of using the stats server to have a top-like realtime monitoring tool (with colors !!!)

2.3.16 Memory usage

Low memory usage is one of the selling point of the whole uWSGI project.

Unfortunately being aggressive with memory by default could (read well: could) lead to some performance problem.

By default the uWSGI rack plugin, calls the ruby GC after every request. If you want to reduce this rate just add the `--rb-gc-freq <n>` option, where n is the number of requests after the GC is called.

If you plan to make benchmarks of uWSGI (or compare it with other solutions) take in account its use of GC.

Ruby can be a memory devourer, so we prefer to be aggressive with memory by default instead of making hello-world benchmarkers happy.

2.3.17 Offloading

The uWSGI offloading subsystem allows you to free your workers as soon as possible when some specific pattern matches and can be delegated to a pure-c thread. Examples are sending static file from the filesystem, transferring data from the network to the client and so on.

Offloading is very complex, but its use is transparent to the end user. If you want to try just add –offload-threads <n> where <n> is the number of threads to spawn (one for cpu is a good value).

When offload threads are enabled, all of the parts that can be optimized will be automatically detected

2.3.18 And now

You should already be able to go in production with such few concepts, but uWSGI is an enormous project with hundreds of features and configurations. If you want to be a better sysadmin, continue reading the full docs.

2.4 Snippets

This is a collection of the most “funny” uses of uWSGI features

2.4.1 X-Sendfile emulation

Even if your frontend proxy/webserver does not support X-Sendfile (or cannot access your static resources) you can emulate it using offloading (your process/thread will delegate the static file serving to offload threads

```
[uwsgi]
...
; load router_static plugin (compiled in by default in monolithic profiles)
plugins = router_static
; spawn 2 offload threads
offload-threads = 2
; files under /private can be safely served
static-safe = /private
; collect the X-Sendfile response header as X_SENDFILE var
collect-header = X-Sendfile X_SENDFILE
; if X_SENDFILE is not empty, pass its value to the "static" routing action (it will automatically us
response-route-if-not = empty:${X_SENDFILE} static:${X_SENDFILE}
```

2.4.2 Force HTTPS

this will force HTTPS for the whole site

```
[uwsgi]
...
; load router_redirect plugin (compiled in by default in monolithic profiles)
plugins = router_redirect
route-if-not = equal:${HTTPS};on redirect-permanent:https://${HTTP_HOST}${REQUEST_URI}
```

and this for /admin

```
[uwsgi]
...
; load router_redirect plugin (compiled in by default in monolithic profiles)
plugins = router_redirect
route = ^/admin goto:https
; stop the chain
route-run = last:

route-label = https
route-if-not = equal:${HTTPS};on redirect-permanent:https://${HTTP_HOST}${REQUEST_URI}
```

2.4.3 Python Auto-reloading (DEVELOPMENT-ONLY !!!)

In production you can monitor file/directory changes for triggering reloads (touch-reload, fs-reload...).

During development having a monitor for all of the loaded/used python modules can be handy. But please use it only during development.

The check is done by a thread that scans the modules list with the specified frequency:

```
[uwsgi]
...
py-autoreload = 2
```

will check for python modules changes every 2 seconds and eventually restart the instance.

Hey, use it only in development...

CHAPTER 3

Table of Contents

3.1 Getting uWSGI

These are the current versions of uWSGI.

Release	Date	Link
Unstable	-	https://github.com/unbit/uwsgi/
Stable/LTS	2014-02-26	http://projects.unbit.it/downloads/uwsgi-2.0.2.tar.gz
Old/LTS	2013-08-23	http://projects.unbit.it/downloads/uwsgi-1.4.10.tar.gz

uWSGI is also available as a package in several OS/distributions.

uWSGI has a really fast development cycle, so packages may not be up to date. Building it requires less than 30 seconds and very few dependencies (only Python interpreter, a C compiler/linker and the libs/headers for your language of choice)

3.2 Installing uWSGI

3.2.1 Installing from a distribution package

See also:

See the *Getting uWSGI* page for a list of known distributions shipping uWSGI.

3.2.2 Installing from source

To build uWSGI you need Python and a C compiler (`gcc` and `clang` are supported). Depending on the languages you wish to support you will need their development headers. On a Debian/Ubuntu system you can install them (and the rest of the infrastructure required to build software) with:

```
apt-get install build-essential python
```

And if you want to build a binary with python/wsgi support (as an example)

```
apt-get install python-dev
```

If you have a variant of *make* available in your system you can simply run *make*. If you do not have *make* (or want to have more control) simply run:

```
python uwsgiconfig.py --build
```

You can also use pip to install uWSGI (it will build a binary with python support).

```
# Install the latest stable release:
pip install uwsgi
# ... or if you want to install the latest LTS (long term support) release,
pip install http://projects.unbit.it/downloads/uwsgi-lts.tar.gz
```

Or you can use ruby gems (it will build a binary with ruby/rack support).

```
# Install the latest stable release:
gem install uwsgi
```

At the end of the build, you will get a report of the enabled features. If something you require is missing, just add the development headers and rerun the build. For example to build uWSGI with ssl and perl regexp support you need libssl-dev and pcre headers.

3.2.3 Alternative build profiles

For historical reasons when you run 'make', uWSGI is built with Python as the only supported language. You can build customized uWSGI servers using build profiles, located in the *buildconf/* directory. You can use a specific profile with:

```
python uwsgiconfig --build <profile>
```

Or you can pass it via an environment variable:

```
UWSGI_PROFILE=lua make
# ... or even ...
UWSGI_PROFILE=gevent pip install uwsgi
```

3.2.4 Modular builds

This is the approach your distribution should follow, and this is the approach you MUST follow if you want to build a commercial service over uWSGI (see below). The vast majority of uWSGI features are available as plugins. Plugins can be loaded using the –plugin option. If you want to give users the maximum amount of flexibility allowing them to use only the minimal amount of resources, just create a modular build. A build profile named "core" is available.

This will build a uWSGi binary without plugins. This is called the "server core". Now you can start building all of the plugins you need. Check the plugins/ directory in the source distribution for a full list.

```
python uwsgiconfig.py --plugin plugins/psgi core
python uwsgiconfig.py --plugin plugins/rack core
python uwsgiconfig.py --plugin plugins/python core
python uwsgiconfig.py --plugin plugins/lua core
python uwsgiconfig.py --plugin plugins/corerouter core
python uwsgiconfig.py --plugin plugins/http core
...
```

Remember to always pass the build profile ('core' in this case) as the third argument.

3.3 The uWSGI build system

- This is updated to 1.9.13

This page describes how the uWSGI build system works and how it can be customized

3.3.1 uwsgiconfig.py

This is the python script aimed at calling the various compile/link stage.

During 2009, when uWSGI guidelines (and mantra) started to be defined, people agreed that autotools, cmake and friends was not loved by a lot of sysadmins. Albeit they are pretty standardized, the amount of packages needed and the incompatibility between them (expecially in the autotools world) was a problem for a project with fast development/evolution where "compile from sources" was, is and very probably will be the best way to get the best from the product. In addition to this the build procedure MUST BE fast (less than 1 minute on entry level x86 is the main rule)

For such a reason, to compile uWSGI you only need to have a c compiler suite (gcc, clang...) and a python interpreter. Someone could argue that perl could have been a better choice, and maybe it is the truth (it is generally installed by default in lot of operating systems), but we decided to stay with python mainly because when uWSGI started it was a python-only application. (Obviously if you want to develop an alternative build system you are free to do it)

The uwsgiconfig.py basically detects the available features in the system and builds a uwsgi binary (and eventually its plugins) using the so called 'build profile'

3.3.2 build profiles

3.3.3 First example

3.3.4 CC and CPP

This 2 environment variables for uwsgiconfig.py to use an alternative C compiler and C preprocessor.

If they are not defined the procedure is the following:

For CC -> try to get the CC config_var from the python binary running uwsgiconfig.py, fallback to 'gcc'

For CPP -> fallback to 'cpp'

As an example, on a system with both gcc and clang you will end with

```
CC=clang CPP=clang-cpp python uwsgiconfig.py --build
```

3.3.5 CPUCOUNT

In the spirit of "easy and fast build even on production systems", uwsgiconfig.py tries to use all of your cpu cores spawning multiple instances of the c compiler (one per-core).

You can override this system using the CPUCOUNT environment variable, forcing the number of detected cpu cores (setting to 1 will disable parallel build).

```
CPUCOUNT=2 python uwsgiconfig.py --build
```

3.3.6 UWSGI_FORCE_REBUILD

3.3.7 Plugins and uwsgiplugin.py

A uWSGI plugin is a shared library exporting the <name>_plugin symbol. Where <name> is the name of the plugin.

As an example the psgi plugin will export the psgi_plugin symbol as well as pypy will export he pypy_plugin symbol and so on.

This symbol is a uwsgi_plugin C struct defining the hooks of the plugin.

When you ask uWSGI to load a plugin it simply calls dlopen() and get the uwsgi_plugin struct via dlsym().

The vast majority of the uWSGI project is developed as a plugin, this ensure a modular approach to configuration and an obviously saner development style.

The sysadmin is free to embed each plugins in the server binary or to build it as an external shared library.

Embedded plugins are defined in the 'embedded_plugins' directive of the build profile. You can add more embedded plugins from command line using UWSGI_EMBED_PLUGINS environment variable (see below).

Instead, if you want to build a plugin as a shared library just run uwsgiconfig.py with the –plugin option

```
python uwsgiconfig.py --plugin plugins/psgi
```

this will build the plugin in plugins/psgi to the psgi_plugin.so file

To specify a build profile when you build a plugin you can pass it as additional argument

```
python uwsgiconfig.py --plugin plugins/psgi mybuildprofile
```

3.3.8 UWSGI_INCLUDES

- this has been added in 1.9.13

On startup, the CPP binary is run to detect default include paths. You can add more paths using the UWSGI_INCLUDES environment variable

```
UWSGI_INCLUDES=/usr/local/include,/opt/dev/include python uwsgiconfig.py --build
```

3.3.9 UWSGI_EMBED_PLUGINS

3.3.10 UWSGI_EMBED_CONFIG

allows embedding the specified .ini file in the server binary (currently Linux only)

On startup the server parses the embedded file as soon as possible.

Custom options defined in the embedded config will be available as standard ones.

3.3.11 UWSGI_BIN_NAME

3.3.12 CFLAGS and LDFLAGS

3.3.13 UWSGICONFIG_* for plugins

3.3.14 libuwsgi.so

3.3.15 uwsgibuild.log

3.3.16 uwsgibuild.lastcflags

3.3.17 cflags and uwsgi.h magic

3.3.18 embedding files

3.3.19 The fake make

3.4 Managing the uWSGI server

See also:

If you are managing multiple apps or a high volume site, take a look at

- *The uWSGI Emperor – multi-app deployment*
- *Zerg mode*
- *uWSGI Subscription Server*

3.4.1 Starting the server

Starting an uWSGI server is the role of the system administrator, like starting the Web server. It should not be the role of the Web server to start the uWSGI server – though you can also do that if it fits your architecture.

How to best start uWSGI services as boot depends on the operating system you use.

On modern systems the following should hold true. On "classic" operating systems you can use `init.d/rc.d` scripts, or tools such as Supervisor, Daemontools or *inetd/xinetd*.

System	Method
Ubuntu	*Running uWSGI via Upstart* (the official `uwsgi` package, available since Ubuntu 12.04 provides an init.d based solution. Read the README.)
Debian	*Running uWSGI via Upstart*
Fedora	*SystemD*
OSX	launchd
Solaris	SMF

3.4.2 Signals for controlling uWSGI

You can instruct uWSGI to write the master process PID to a file with the `pidfile` option.

The uWSGI server responds to the following signals.

Signal	Description	Convenience command
SIGHUP	gracefully reload all the workers and the master process	–reload
SIGTERM	brutally reload all the workers and the master process	
SIGINT	immediately kill the entire uWSGI stack	–stop
SIGQUIT	immediately kill the entire uWSGI stack	
SIGUSR1	print statistics	
SIGUSR2	print worker status or wakeup the spooler	
SIGURG	restore a snapshot	
SIGTSTP	pause/suspend/resume an instance	
SIGWINCH	wakeup a worker blocked in a syscall (internal use)	

3.4.3 Reloading the server

When running with the `master` process mode, the uWSGI server can be gracefully restarted without closing the main sockets.

This functionality allows you patch/upgrade the uWSGI server without closing the connection with the web server and losing a single request.

When you send the *SIGHUP* to the master process it will try to gracefully stop all the workers, waiting for the completion of any currently running requests.

Then it closes all the eventually opened file descriptor not related to uWSGI.

Lastly, it binary patches (using `execve()`) the uWSGI process image with a new one, inheriting all of the previous file descriptors.

The server will know that it is a reloaded instance and will skip all the sockets initialization, reusing the previous ones.

Note: Sending the *SIGTERM* signal will obtain the same result reload-wise but will not wait for the completion of running requests.

There are several ways to make uWSGI gracefully restart.

```
# using kill to send the signal
kill -HUP `cat /tmp/project-master.pid`
# or the convenience option --reload
uwsgi --reload /tmp/project-master.pid
# or if uwsgi was started with touch-reload=/tmp/somefile
touch /tmp/somefile
```

Or from your application, in Python:

```
uwsgi.reload()
```

Or in Ruby,

```
UWSGI.reload
```

3.4.4 Stopping the server

If you have the uWSGI process running in the foreground for some reason, you can just hit CTRL+C to kill it off.

When dealing with background processes, you'll need to use the master pidfile again. The SIGINT signal will kill uWSGI.

```
kill -INT `cat /tmp/project-master.pid`
# or for convenience...
uwsgi --stop /tmp/project-master.pid
```

3.4.5 The Master FIFO

Starting from uWSGI 1.9.17 a new management system has been added using unix named pipes (fifo): *The Master FIFO*

3.5 Supported languages and platforms

Technology	Available since	Notes	Status
Python	0.9.1	The first available plugin, supports WSGI (**PEP 333**, **PEP 3333**), Web3 (from version 0.9.7-dev) and Pump (from 0.9.8.4). Works with `Virtualenv`, multiple Python interpreters, `Python3` and has unique features like *Aliasing Python modules*, `DynamicVirtualenv` and *uGreen – uWSGI Green Threads*. A module exporting handy *decorators* for the uWSGI API is available in the source distribution. PyPy *is supported* since 1.3. The *Python Tracebacker* was added in 1.3.	Stable, 100% uWSGI API support
Lua	0.9.5	Supports `LuaWSAPI`, coroutines and threads	Stable, 60% uWSGI API support
Perl	0.9.5	*uWSGI Perl support (PSGI)* (PSGI) support. Multiple interpreters, threading and async modes supported	Stable, 60% uWSGI API support
Ruby	0.9.7-dev	*Ruby support* support. A loop engine for `Ruby 1.9 fibers` is available as well as a handy `DSL` module.	Stable, 80% uWSGI API support
Integrating uWSGI with Erlang	0.9.5	Allows message exchanging between uWSGI and Erlang nodes.	Stable, no uWSGI API support
Running CGI scripts on uWSGI	1.0-dev	Run CGI scripts	Stable, no uWSGI API support
Running PHP scripts in uWSGI	1.0-dev	Run PHP scripts	Stable from 1.1, 5% uWSGI API support
uWSGI Go support (1.4 only)	1.4-dev	Allows integration with the Go language	15% uWSGI API support
JVM in the uWSGI server (updated to 1.9)	1.9-dev	Allows integration between uWSGI and the Java Virtual Machine *JWSGI* and *Clojure/Ring* handlers are available.	Stable
The Mono ASP.NET plugin	0.9.7-dev	Allows integration between uWSGI and Mono, and execution of ASP.NET applications.	Stable
V8	1.9.4	Allows integration between uWSGI and javascript V8.	Early stage of development

3.6 Supported Platforms/Systems

This is the list of officially supported operating systems and platforms.

- Linux 2.6/3.x
- FreeBSD >= 7
- NetBSD
- OpenBSD

- DragonFlyBSD
- Windows Cygwin
- Mac OSX
- Solaris >= 10
- NexentaOS
- SmartOS
- OpenSolaris
- OpenIndiana
- OmniOS
- Debian/kFreeBSD
- GNU/Hurd

3.7 Web server integration

uWSGI supports several methods of integrating with web servers. It is also capable of serving HTTP requests by itself.

3.7.1 Cherokee

See also:

Cherokee support

The Cherokee webserver officially supports uWSGI. Cherokee is fast and lightweight, has a beautiful admin interface and a great community. Their support for uWSGI has been awesome since the beginning and we recommend its use in most situations. The userbase of the Cherokee uWSGI handler is probably the biggest of all. The Cherokee uWSGI handler is commercially supported by Unbit.

3.7.2 Nginx

See also:

Nginx support

The uWSGI module is included in the official Nginx distribution since version 0.8.40. A version supporting Nginx 0.7.x is maintained in the uWSGI package.

This is a stable handler commercially supported by Unbit.

3.7.3 Apache

See also:

Apache support

The Apache2 *mod_uwsgi* module was the first web server integration module developed for uWSGI. It is stable but could be better integrated with the Apache API.

It is commercially supported by Unbit.

Since uWSGI 0.9.6-dev a second Apache2 module called *mod_Ruwsgi* is included. It's more Apache API friendly. *mod_Ruwsgi is not commercially supported by Unbit.*

During the 1.2 development cycle, another module called *mod_proxy_uwsgi* has been added. In the near future this should be the best choice for Apache based deployments.

3.7.4 Mongrel2

See also:

Attaching uWSGI to Mongrel2

Support for the Mongrel2 Project has been available since 0.9.8-dev via the `ZeroMQ` protocol plugin.

In our tests Mongrel2 survived practically all of the loads we sent.

Very good and solid project. Try it :)

3.7.5 Lighttpd

This module is the latest developed, but its inclusion in the official Lighttpd distribution has been rejected, as the main author considers the *uwsgi protocol* a "reinventing the wheel" technology while suggesting a FastCGI approach. We respect this position. The module will continue to reside in the uWSGI source tree, but it is currently unmaintained.

There is currently no commercial support for this handler. We consider this module "experimental".

3.7.6 Twisted

This is a "commodity" handler, useful mainly for testing applications without installing a full web server. If you want to develop an uWSGI server, look at this module. `Twisted`.

3.7.7 Tomcat

The included servlet can be used to forward requests from Tomcat to the uWSGI server. It is stable, but currently lacks documentation.

There is currently no commercial support for this handler.

3.7.8 CGI

The CGI handlers are for "lazy" installations. Their use in production environments is discouraged.

3.8 Frequently Asked Questions (FAQ)

3.8.1 Why should I choose uWSGI?

Because you can! :) uWSGI wants to be a complete web application deployment solution with batteries included:

- `ProcessManagement`
- Management of long-running tasks
- *uWSGI RPC Stack*

- `Clustering`
- `LoadBalancing`
- `Monitoring`
- `ResourceLimiting`

... and many other annoying everyday tasks that you'd have to delegate to external scripts and manual sysadmin tasks.

If you are searching for a simple server for your WSGI, PSGI or Rack app, uWSGI may not be for you. Though, if you are building an app which needs to be rock solid, fast, and easy to distribute and optimize for various loads, you will most likely find yourself needing uWSGI.

The best definition for uWSGI is "Swiss Army Knife for your network applications".

3.8.2 What about the protocol?

The uwsgi (all lowercase) protocol is derived from SCGI but with binary string length representations and a 4-byte header that includes the size of the var block (16 bit length) and a couple of general-purpose bytes. We are not reinventing the wheel. Binary management is much easier and cheaper than string parsing, and every single bit of power is required for our projects. If you need proof, look at the *official protocol documentation* and you will understand why a new protocol was needed. Obviously, you are free to use the other supported protocols. Remember, if you cannot use uWSGI in some scenario, it is a uWSGI bug.

3.8.3 Can I use it in cluster environments?

Yes, this is one of the main features of the uWSGI stack. You can have multiple instances bound on different servers, and using the load balancing facilities of your webserver/proxy/router you can distribute your load. Systems like *uWSGI RPC Stack* allows you to fast call functions on remote nodes, and *The uWSGI Legion subsystem* allows you to elect a master in a multi-node setup.

3.8.4 So, why all those timeout configuration flags?

Choosing sane timeouts is the key to high availability. Do not trust network applications that do not permit you to choose a timeout.

3.8.5 I need help! What do I do?

Post a message on the uWSGI mailing list including your

- Operating system version
- CPU architecture
- Webserver used (if any)
- uWSGI version
- uWSGI command line or config files

You should add the *–show-config* option and post the output in the message. It will be very useful for finding out just what's wrong with your uWSGI. You can also `rebuild uWSGI` with debug symbols and run it under a debugger like *gdb*.

uWSGI is an enormous project with hundreds of options. You should be prepared that not everything will go right at the first shot. Ask for help, ask for help and ask for help. If you are frustrated, do not waste time blaming and ranting - instead simply join the list and ask for help. This is open source, if you only rant you are doing nothing useful.

3.8.6 I am not a sysadmin, nor a UNIX guru. Can I use uWSGI?

That's a good question :) But sadly there is no simple answer. uWSGI has not been developed with simplicity in mind, but with versatility. You can try it by starting with one of the quickstarts and if you have problems, simply ask for help in the list or on the IRC channel.

3.8.7 How can I buy commercial support for my company?

Send an email to info at unbit.it with the word "uWSGI" in the subject. The email you send should include your company information and your specific request. We will reply as soon as possible.

3.8.8 Will this allow me to run my awesome apps on my ancient close-minded ISP?

Probably not. The uWSGI server requires a modern platform/environment.

3.8.9 Where are the benchmarks?

Sorry, we only do "official" benchmarks for regression testing. If benchmarks are very important to you, you can search on the mailing list, make your own benchmarks or search on Google. uWSGI gives precedence to machine health, so do not expect that your *ab* test with an unrealistic number of concurrent connections will be managed flawlessly without tuning. Some socket and networking knowledge is required if you want to make a valid benchmark (and avoid geek rage in your blog comments ;). Also remember that uWSGI can be run in various modes, so avoid comparing it configured in preforking mode with another server in non-blocking/async mode if you do not want to look ridiculous.

Note: If you see your tests failing at higher concurrency rates you are probably hitting your OS socket backlog queue limit (maximum of 128 slots on Linux, tunable via */proc/sys/net/somaxconn* and */proc/sys/net/ipv4/tcp_max_syn_backlog* for TCP sockets).

You can set this value in uWSGI with the *listen* configuration option.

3.8.10 Ha! Server XXX is faster than uWSGI! Take that!

As already stated uWSGI is not a silver bullet, it is not meant to be liked by the whole world and it is obviously not the fastest server out there. It is a piece of software following an "approach" to problems you may not like or that you may conversely love. The approach taken will work better for certain cases than others, and each application should be analyzed on it's own merits using appropriate and accruate real-world benchmarks.

3.8.11 What is 'Harakiri mode'?

At Unbit we host hundreds of unreliable web apps on our servers. All of them run on hardly constrained (at kernel level) environments where having processes block due to an implementation error will result on taking down an entire site. The harakiri mode has two operational modes:

- one that we define as "raw and a bit unreliable" (used for simple setup without a process manager)

- and another one that we define as "reliable" that depends on the presence of the uWSGI process manager (see `ProcessManagement`).

The first one sets a simple alarm at the start of every request. If the process gets a *SIGALRM* signal, it terminates itself. We call this unreliable, because your app or some module you use could overwrite or simply cancel the alarm with a simple call to *alarm()*.

The second one uses a master process shared memory area (via *mmap*) that maintains statistics on every worker in the pool. At the start of every request, the worker sets a timestamp representing the time after the process will be killed in its dedicated area. This timestamp is zeroed after every successful request. If the master process finds a worker with a timestamp in the past it will mercilessly kill it.

3.8.12 Will my app run faster with uWSGI?

It's unlikely. The biggest bottleneck in web app deployment is the application itself. If you want a faster environment, optimize your code or use techniques such as clustering or caching. We say that uWSGI is fast because it introduces a very little overhead in the deployment structure.

3.8.13 What are the most important options for performance and robustness in the uWSGI environment?

By default, uWSGI is configured with sane "almost-good-for-all" values. But if and when things start going wild, tuning is a must.

- Increasing (or decreasing) timeout is important, as is modifying the socket listen queue size.
- Think about threading. If you do not need threads, do not enable them.
- If you are running only a single application you can disable multiple interpreters.
- Always remember to enable the master process in production environments. See `ProcessManagement`.
- Adding workers does not mean "increasing performance", so choose a good value for the *workers* option based on the nature of your app (IO bound, CPU bound, IO waiting...)

3.8.14 Why not simply use HTTP as the protocol?

A good question with a simple answer: HTTP parsing is slow, really slow. Why should we do a complex task twice? The web server has already parsed the request! The *uwsgi protocol* is very simple to parse for a machine, while HTTP is very easy to parse for a human. As soon as humans are being used as servers, we will abandon the uwsgi protocol in favor of the HTTP protocol. All this said, you can use uWSGI via *Native HTTP support*, `FastCGI`, `ZeroMQ` and other protocols as well.

3.8.15 Why do you support multiple methods of configuration?

System administration is all about skills and taste. uWSGI tries to give sysadmins as much choice as possible for integration with whatever infrastructure already available. Having multiple methods of configuration is just one way we achieve this.

3.8.16 What is the best webserver handler?

See *Web server integration*.

3.9 Things to know (best practices and "issues")

- The `http` and `http-socket` options are entirely different beasts. .. seealso:: *Native HTTP support*
- By default sending the `SIGTERM` signal to uWSGI means "brutally reload the stack" while the convention is to shut an application down on `SIGTERM`. To shutdown uWSGI use `SIGINT` or `SIGQUIT` instead. If you absolutely can not live with uWSGI being so disrespectful towards `SIGTERM`, by all means enable the `die-on-term` option.
- If you plan to host multiple applications do yourself a favor and check the *The uWSGI Emperor – multi-app deployment*.
- Always use uwsgitop, through *The uWSGI Stats Server* or something similar to monitor your apps' health.
- uWSGI can include features in the core or as loadable plugins. uWSGI packages supplied with OS distributions tend to be modular. In such setups, be sure to load the plugins you require with the `plugins` option. A good symptom to recognize an unloaded plugin is messages like "Unavailable modifier requested" in your logs. If you are using distribution supplied packages, double check you have installed the plugin for your language of choice.
- Config files support a limited form of inheritance, variables, if constructs and simple cycles. Check the *Configuration logic* and *How uWSGI parses config files* pages.
- To route requests to a specific plugin, the webserver needs to pass a magic number known as a modifier to the uWSGI instances. By default this number is set to 0, which is mapped to Python. As an example, routing a request to a `PSGI` app requires you to set the modifier to `5` - or optionally to load the PSGI plugin as modifier `0`. (This will mean that all modifierless requests will be considered Perl.)
- There is no magic rule for setting the number of processes or threads to use. It is very much application and system dependent. Simple math like `processes = 2 * cpucores` will not be enough. You need to experiment with various setups and be prepared constantly monitor your apps. `uwsgitop` could be a great tool to find the best values.
- If an HTTP request has a body (like a POST request generated by a form), you *have* to read (consume) it in your application. If you do not do this, the communication socket with your webserver may be clobbered. If you are lazy you can use the `post-buffering` option that will automatically read data for you. For `Rack` applications this is automatically enabled.
- Always check the memory usage of your apps. The `memory-report` option could be your best friend.
- If you plan to use UNIX sockets (as opposed to TCP), remember they are standard filesystem objects. This means they have permissions and as such your webserver must have write access to them.
- Common sense: do not run uWSGI instances as root. You can start your uWSGIs as root, but be sure to drop privileges with the `uid` and `gid` options.
- uWSGI tries to (ab)use the Copy On Write semantics of the fork() call whenever possible. By default it will fork after having loaded your applications to share as much of their memory as possible. If this behavior is undesirable for some reason, use the `lazy` option. This will instruct uWSGI to load the applications after each worker's `fork()`. Lazy mode changes the way graceful reloading works: instead of reloading the whole instance, each worker is reloaded in chain. If you want "lazy app loading", but want to maintain the standard uWSGI reloading behaviour, starting from 1.3 you can use the `lazy-apps` option.
- By default the Python plugin does not initialize the GIL. This means your app-generated threads will not run. If you need threads, remember to enable them with `enable-threads`. Running uWSGI in multithreading mode (with the `threads` options) will automatically enable threading support. This "strange" default behaviour is for performance reasons, no shame in that.
- If you spawn a new process during a request it will inherit the file descriptors of the worker spawning it - including the socket connected with the webserver/router. If you do not want this behaviour set the `close-on-exec`

option.

- The Ruby garbage collector is configured by default to run after every request. This is an aggressive policy that may slow down your apps a bit – but CPU resources are cheaper than memory, and especially cheaper than running out of memory. To tune this frequency use the `ruby-gc <freq>` option.
- On OpenBSD, NetBSD and FreeBSD < 9, SysV IPC semaphores are used as the locking subsystem. These operating systems tend to limit the number of allocable semaphores to fairly small values. You should raise the default limits if you plan to run more than one uWSGI instance. FreeBSD 9 has POSIX semaphores, so you do not need to bother with that.
- Do not build plugins using a different config file than used to build the uWSGI binary itself – unless you like pain or know *exactly* what you are doing.
- By default uWSGI allocates a very small buffer (4096 bytes) for the headers of each request. If you start receiving "invalid request block size" in your logs, it could mean you need a bigger buffer. Increase it (up to 65535) with the `buffer-size` option. .. note:: If you receive '21573' as the request block size in your logs, it could mean you are using the HTTP protocol to speak with an instance speaking the uwsgi protocol. Don't do this.
- If your (Linux) server seems to have lots of idle workers, but performance is still sub-par, you may want to look at the value of the `ip_conntrack_max` system variable (`/proc/sys/net/ipv4/ip_conntrack_max`) and increase it to see if it helps.
- Some Linux distributions (read: Debian Etch 4) make a mix of newer kernels with very old userspace. This kind of combination can make the uWSGI build system spit out errors (most notably on `unshare()`, pthread locking, `inotify`...). You can force uWSGI to configure itself for an older system prefixing the 'make' (or whatever way you use to build it) with `CFLAGS="-DOBSOLETE_LINUX_KERNEL"`
- By default the stdin is remapped to /dev/null on startup. If you need a valid stdin (for debugging, piping and so on) add –honour-stdin
- You can easily add non-existent options to your config files (as placeholder, custom options, or app-related configuration items). This is a really handy feature, but can lead to headaches on typos. The strict mode (–strict) will disable this feature, and only valid uWSGI options are tolerated.
- Some plugin (most notably python and perl) has code auto-reloading facilities. Albeit they are very appealing, you MUST use them only under development as they are really heavyweight. For example the python –py-autoreload option will scan your whole module tree at every check cycle.

3.10 Configuring uWSGI

uWSGI can be configured using several different methods. All configuration methods may be mixed and matched in the same invocation of uWSGI.

Note: Some of the configuration methods may require a specific plugin (ie. sqlite and ldap).

See also:

Configuration logic

The configuration system is unified, so each command line option maps 1:1 with entries in the config files.

Example:

```
uwsgi --http-socket :9090 --psgi myapp.pl
```

can be written as

```
[uwsgi]
http-socket = :9090
psgi = myapp.pl
```

3.10.1 Loading configuration files

uWSGI supports loading configuration files over several methods other than simple disk files:

```
uwsgi --ini http://uwsgi.it/configs/myapp.ini # HTTP
uwsgi --xml - # standard input
uwsgi --yaml fd://0 # file descriptor
uwsgi --json 'exec://nc 192.168.11.2:33000' # arbitrary executable
```

Note: More esoteric file sources, such as the *Emperor*, embedded configuration (in two flavors), dynamic library symbols and ELF sections could also be used.

3.10.2 Magic variables

uWSGI configuration files can include "magic" variables, prefixed with a percent sign. Currently the following magic variables (you can access them in Python via `uwsgi.magic_table`) are defined.

%v	the vassals directory (pwd)
%V	the uWSGI version
%h	the hostname
%o	the original config filename, as specified on the command line
%O	same as %o but refer to the first non-template config file (version 1.9.18)
%p	the absolute path of the configuration file
%P	same as %p but refer to the first non-template config file (version 1.9.18)
%s	the filename of the configuration file
%S	same as %s but refer to the first non-template config file (version 1.9.18)
%d	the absolute path of the directory containing the configuration file
%D	same as %d but refer to the first non-template config file (version 1.9.18)
%e	the extension of the configuration file
%E	same as %e but refer to the first non-template config file (version 1.9.18)
%n	the filename without extension
%N	same as %n but refer to the first non-template config file (version 1.9.18)
%c	the name of the directory containing the config file (version 1.3+)
%C	same as %c but refer to the first non-template config file (version 1.9.18)
%t	unix time (in seconds, gathered at instance startup) (version 1.9.20-dev+)
%T	unix time (in microseconds, gathered at instance startup) (version 1.9.20-dev+)
%x	the current section identifier, eg. *config.ini:section* (version 1.9-dev+)
%X	same as %x but refer to the first non-template config file (version 1.9.18)
%i	inode number of the file (version 2.0.1)
%I	same as %i but refer to the first non-template config file
%0..%9	a specific component of the full path of the directory containing the config file (version 1.3+)
%[	ANSI escape "\033" (useful for printing colors)
%k	detected cpu cores (version 1.9.20-dev+)
%u	uid of the user running the process (version 2.0)
%U	username (if available, otherwise fallback to uid) of the user running the process (version 2.0)
%g	gid of the user running the process (version 2.0)
%G	group name (if available, otherwise fallback to gid) of the user running the process (version 2.0)

Continued on next page

Table 3.1 – continued from previous page

%j	HEX representation of the djb33x hash of the full config path
%J	same as %j but refer to the first non-template config file

Note that most of these refer to the file they appear in, even if that file is included from another file.

An exception are most of the uppercase versions, which refer to the first non-template config file loaded. This means the first config file not loaded through `--include` or `--inherit`, but through for example `--ini`, `--yaml` or `--config`. These are intended to use with the emperor, to refer to the actual vassal config file instead of templates included with `--vassals-include` or `--vassals-inherit`.

For example, here's `funnyapp.ini`.

```
[uwsgi]
socket = /tmp/%n.sock
module = werkzeug.testapp:test_app
processes = 4
master = 1
```

`%n` will be replaced with the name of the config file, sans extension, so the result in this case will be

```
[uwsgi]
socket = /tmp/funnyapp.sock
module = werkzeug.testapp:test_app
processes = 4
master = 1
```

3.10.3 Placeholders

Placeholders are custom magic variables defined during configuration time by setting a new configuration variable of your own devising.

```
[uwsgi]
; These are placeholders...
my_funny_domain = uwsgi.it
set-ph = max_customer_address_space = 64
set-placeholder = customers_base_dir = /var/www
; And these aren't.
socket = /tmp/sockets/%(my_funny_domain).sock
chdir = %(customers_base_dir)/%(my_funny_domain)
limit-as = %(max_customer_address_space)
```

Placeholders can be assigned directly, or using the `set-placeholder` / `set-ph` option. These latter options can be useful to:

- Make it more explicit that you're setting placeholders instead of regular options.
- Set options on the commandline, since unknown options like `--foo=bar` are rejected but `--set-placeholder foo=bar` is ok.
- Set placeholders when strict mode is enabled.

Placeholders are accessible, like any uWSGI option, in your application code via `uwsgi.opt`.

```
import uwsgi
print uwsgi.opt['customers_base_dir']
```

This feature can be (ab)used to reduce the number of configuration files required by your application.

Similarly, contents of evironment variables and external text files can be included using the *@(file_name)* and *$(ENV_VAR)* syntax. See also *How uWSGI parses config files*.

3.10.4 Placeholders math (from uWSGI 1.9.20-dev)

You can apply math formulas to placeholders using this special syntax:

```
[uwsgi]
foo = 17
bar = 30
; total will be 50
total = %(foo + bar + 3)
```

Remember to not miss spaces between operations.

Operations are executed in a pipeline (not in common math style):

```
[uwsgi]
foo = 17
bar = 30
total = %(foo + bar + 3 * 2)
```

'total' will be evaluated as 100:

(((foo + bar) + 3) * 2)

Incremental and decremental shortcuts are available

```
[uwsgi]
foo = 29
; remember the space !!!
bar = %(foo ++)
```

bar will be 30

If you do not specify an operation between two items, 'string concatenation' is assumed:

```
[uwsgi]
foo = 2
bar = 9
; remember the space !!!
bar = %(foo bar ++)
```

the first two items will be evaluated as '29' (not 11 as no math operation has been specified)

3.10.5 Command line arguments

Example:

```
uwsgi --socket /tmp/uwsgi.sock --socket 127.0.0.1:8000 --master --workers 3
```

3.10.6 Environment variables

When passed as environment variables, options are capitalized and prefixed with *UWSGI_*, and dashes are substituted with underscores.

Note: Several values for the same configuration variable are not supported with this method.

Example:

```
UWSGI_SOCKET=127.0.0.1 UWSGI_MASTER=1 UWSGI_WORKERS=3 uwsgi
```

3.10.7 INI files

.INI files are a standard de-facto configuration format used by many applications. It consists of `[section]``s and ``key=value` pairs.

An example uWSGI INI configuration:

```
[uwsgi]
socket = /tmp/uwsgi.sock
socket = 127.0.0.1:8000
workers = 3
master = true
```

By default, uWSGI uses the `[uwsgi]` section, but you can specify another section name while loading the INI file with the syntax `filename:section`, that is:

```
uwsgi --ini myconf.ini:app1
```

Alternatively, you can load another section from the same file by omitting the filename and specifying just the section name. Note that technically, this loads the named section from the last .ini file loaded instead of the current one, so be careful when including other files.

```
[uwsgi]
# This will load the app1 section below
ini = :app1
# This will load the defaults.ini file
ini = defaults.ini
# This will load the app2 section from the defaults.ini file!
ini = :app2

[app1]
plugin = rack

[app2]
plugin = php
```

- Whitespace is insignificant within lines.
- Lines starting with a semicolon (`;`) or a hash/octothorpe (`#`) are ignored as comments.
- Boolean values may be set without the value part. Simply `master` is thus equivalent to `master=true`. This may not be compatible with other INI parsers such as `paste.deploy`.
- For convenience, uWSGI recognizes bare `.ini` arguments specially, so the invocation `uwsgi myconf.ini` is equal to `uwsgi --ini myconf.ini`.

3.10.8 XML files

The root node should be `<uwsgi>` and option values text nodes.

An example:

```
<uwsgi>
  <socket>/tmp/uwsgi.sock</socket>
  <socket>127.0.0.1:8000</socket>
  <master/>
  <workers>3</workers>
</uwsgi>
```

You can also have multiple `<uwsgi>` stanzas in your file, marked with different `id` attributes. To choose the stanza to use, specify its id after the filename in the `xml` option, using a colon as a separator. When using this *id* mode, the root node of the file may be anything you like. This will allow you to embed `uwsgi` configuration nodes in other XML files.

```
<i-love-xml>
  <uwsgi id="turbogears"><socket>/tmp/tg.sock</socket></uwsgi>
  <uwsgi id="django"><socket>/tmp/django.sock</socket></uwsgi>
</i-love-xml>
```

- Boolean values may be set without a text value.
- For convenience, uWSGI recognizes bare `.xml` arguments specially, so the invocation `uwsgi myconf.xml` is equal to `uwsgi --xml myconf.xml`.

3.10.9 JSON files

The JSON file should represent an object with one key-value pair, the key being *"uwsgi"* and the value an object of configuration variables. Native JSON lists, booleans and numbers are supported.

An example:

```
{"uwsgi": {
  "socket": ["/tmp/uwsgi.sock", "127.0.0.1:8000"],
  "master": true,
  "workers": 3
}}
```

Again, a named section can be loaded using a colon after the filename.

```
{"app1": {
  "plugin": "rack"
}, "app2": {
  "plugin": "php"
}}
```

And then load this using:

```
uwsgi --json myconf.json:app2
```

Note: The Jansson library is required during uWSGI build time to enable JSON support. By default the presence of the library will be auto-detected and JSON support will be automatically enabled, but you can force JSON support to be enabled or disabled by editing your build configuration.

See also:

Installing uWSGI

3.10.10 YAML files

The root element should be *uwsgi*. Boolean options may be set as *true* or *1*.

An example:

```
uwsgi:
  socket: /tmp/uwsgi.sock
  socket: 127.0.0.1:8000
  master: 1
  workers: 3
```

Again, a named section can be loaded using a colon after the filename.

```
app1:
  plugin: rack
app2:
  plugin: php
```

And then load this using:

```
uwsgi --yaml myconf.yaml:app2
```

3.10.11 SQLite configuration

Note: Under construction.

3.10.12 LDAP configuration

LDAP is a flexible way to centralize configuration of large clusters of uWSGI servers. Configuring it is a complex topic. See *Configuring uWSGI with LDAP* for more information.

3.11 Fallback configuration

(available from 1.9.15-dev)

If you need a "reset to factory defaults", or "show a welcome page if the user has made mess with its config" scenario, fallback configuration is your silver bullet

3.11.1 Simple case

A very common problem is screwing-up the port on which the instance is listening.

To emulate this kind of error we try to bind on port 80 as unprivileged user:

```
uwsgi --uid 1000 --http-socket :80
```

uWSGI will exit with:

```
bind(): Permission denied [core/socket.c line 755]
```

Internally (from the kernel point of view) the instance exited with status 1

Now we want to allow the instance to automatically bind on port 8080 when the user supplied config fails.

Let's define a fallback config (you can save it as safe.ini):

```
[uwsgi]
print = Hello i am the fallback config !!!
http-socket = :8080
wsgi-file = welcomeapp.wsgi
```

Now we can re-run the (broken) instance:

```
uwsgi --fallback-config safe.ini --uid 1000 --http-socket :80
```

Your error will be now something like:

```
bind(): Permission denied [core/socket.c line 755]
Thu Jul 25 21:55:39 2013 - !!! /home/roberto/uwsgi/uwsgi (pid: 7409) exited with status 1 !!!
Thu Jul 25 21:55:39 2013 - !!! Fallback config to safe.ini !!!
[uWSGI] getting INI configuration from safe.ini
*** Starting uWSGI 1.9.15-dev-a0cb71c (64bit) on [Thu Jul 25 21:55:39 2013] ***
...
```

as you can see the instance has detected the exit code 1, and binary patched itself with a new one (without changing the pid, or calling fork())

3.11.2 Broken apps

Another common problem is the inhability to load an application, but instead bringing down the whole site we want to load an alternate application:

```
uwsgi --fallback-config safe.ini --need-app --http-socket :8080 --wsgi-file brokenapp.py
```

Here the key is –need-app. It will call exit(1) if the instance has not been able to load at least one application.

3.11.3 Multiple fallback levels

Your fallback config file can specify a fallback-config directive too, allowing multiple fallback levels. BEWARE OF LOOPS !!!3

3.11.4 How it works

The objective is catching the exit code of a process before the process itself is destroyed (we do not want to call another fork(), or destroy already opened file descriptors)

uWSGI makes heavy usage of atexit() hooks, so we only need to register the fallback handler as the first one (hooks are executed in reverse order).

In addition to this we need to get the exit code in our atexit() hook, something is not supported by default (the on_exit() function is now deprecated).

The solution is "patching" exit(x) with uwsgi_exit(x) that is a simple wrapper setting uwsgi.last_exit_code memory pointer.

Now the hook only needs to check for uwsgi.last_exit_code == 1 and eventually execve() the binary again passing the fallback config to it

```
char *argv[3];
argv[0] = uwsgi.binary_path;
argv[1] = uwsgi.fallback_config;
argv[2] = NULL;
execvp(uwsgi.binary_path, argv);
```

3.11.5 Notes

Try to place –fallback-config as soon as possibile in your config tree. The various config parsers may fail (calling exit(1)) before the fallback file is registered

3.12 Configuration logic

Starting from 1.1 certain logic constructs are available.

The following statements are currently supported:

- `for .. endfor`
- `if-dir / if-not-dir`
- `if-env / if-not-env`
- `if-exists / if-not-exists`
- `if-file / if-not-file`
- `if-opt / if-not-opt`
- `if-reload / if-not-reload` – undocumented

Each of these statements exports a context value you can access with the special placeholder `%(_)`. For example, the "for" statement sets `%(_)` to the current iterated value.

Warning: Recursive logic is not supported and will cause uWSGI to promptly exit.

3.12.1 for

For iterates over space-separated strings. The following three code blocks are equivalent.

```
[uwsgi]
master = true
; iterate over a list of ports
for = 3031 3032 3033 3034 3035
socket = 127.0.0.1:%(_)
endfor =
module = helloworld
```

```
<uwsgi>
  <master/>
  <for>3031 3032 3033 3034 3035</for>
    <socket>127.0.0.1:%(_)</socket>
  <endfor/>
  <module>helloworld</module>
</uwsgi>
```

```
uwsgi --for="3031 3032 3033 3034 3035" --socket="127.0.0.1:%(_)" --endfor --module helloworld
```

Note that the for-loop is applied to each line inside the block separately, not to the block as a whole. For example, this:

```
[uwsgi]
for = a b c
socket = /var/run/%(_).socket
http-socket = /var/run/%(_)-http.socket
endfor =
```

is expanded to:

```
[uwsgi]
socket = /var/run/a.socket
socket = /var/run/b.socket
socket = /var/run/c.socket
http-socket = /var/run/a-http.socket
http-socket = /var/run/b-http.socket
http-socket = /var/run/c-http.socket
```

3.12.2 if-env

Check if an environment variable is defined, putting its value in the context placeholder.

```
[uwsgi]
if-env = PATH
print = Your path is %(_)
check-static = /var/www
endif =
socket = :3031
```

3.12.3 if-exists

Check for the existence of a file or directory. The context placeholder is set to the filename found.

```
[uwsgi]
http = :9090
; redirect all requests if a file exists
if-exists = /tmp/maintainance.txt
route = .* redirect:/offline
endif =
```

Note: The above example uses *uWSGI internal routing*.

3.12.4 if-file

Check if the given path exists and is a regular file. The context placeholder is set to the filename found.

```
<uwsgi>
  <plugins>python</plugins>
  <http-socket>:8080</http-socket>
  <if-file>settings.py</if-file>
    <module>django.core.handlers.wsgi:WSGIHandler()</module>
```

```
  <endif/>
</uwsgi>
```

3.12.5 if-dir

Check if the given path exists and is a directory. The context placeholder is set to the filename found.

```
uwsgi:
  socket: 4040
  processes: 2
  if-file: config.ru
  rack: %(_)
  endif:
```

3.12.6 if-opt

Check if the given option is set, or has a given value. The context placeholder is set to actual value of the option reference.

To check if an option was set, pass just the option name to `if-opt`.

```
uwsgi:
  cheaper: 3
  if-opt: cheaper
  print: Running in cheaper mode, with initially %(_) processes
  endif:
```

To check if an option was set to a specific value, pass `option-name=value` to `if-opt`.

```
uwsgi:
  # Set busyness parameters if it was chosen
  if-opt: cheaper-algo=busyness
  cheaper-busyness-max: 25
  cheaper-busyness-min: 10
  endif:
```

Due to the way uWSGI parses its configs, you can only refer to options that uWSGI has previously seen. In particular, this means:

- Only options that are set above the `if-opt` option are taken into account. This includes any options set by previous `include` (or type specific includes like `ini`) options, but does not include options set by previous `inherit` options).
- `if-opt` is processed after expanding magic variables, but before expanding placeholders and other variables. So if you use `if-opt` to compare the value of an option, check against the value as stated in the config file, with only the magic variables filled in.

  If you use the context placeholder `%(_)` inside the `if-opt` block, you should be ok: any placeholders will later be expanded.
- If an option is specified multiple times, only the value of the first one will be seen by `if-opt`.
- Only explicitely set values will be seen, not implicit defaults.

See also:

How uWSGI parses config files

3.13 Configuration Options

uWSGI and the various plugins it consists of is almost infinitely configurable.

There's an exhaustive and exhausting list of all options below. Take a deep breath and don't panic – the list below is long, but you don't need to know everything to start using uWSGI.

3.13.1 Networking/sockets

socket, uwsgi-socket

Argument: string

Bind to the specified socket using default protocol (see *protocol*).

This option may be set with `-s` from the command line.

http-socket

Argument: string

Bind to the specified socket using HTTP.

fastcgi-socket

Argument: string

Bind to the specified socket using FastCGI.

protocol, socket-protocol

Argument: string

Force the specified protocol (*uwsgi*, *http*, *fastcgi*) for default sockets.

shared-socket

Argument: string

Create a shared socket for advanced jailing or IPC purposes.

Advanced option for plugin writers or special needs. Allows you to create a socket early in the server's startup and use it after privileges drop or jailing. This can be used to bind to privileged (<1024) ports.

listen

Argument: number **Default:** 100

Set the socket listen queue size.

This option may be set with `-l` from the command line.

Every socket has an associated queue where request will be put waiting for a process to became ready to accept them. When this queue is full, requests will be rejected.

The maximum value is system/kernel dependent.

abstract-socket

Argument: no argument

Force UNIX socket into abstract mode (Linux only).

chmod-socket

Argument: optional string

Chmod socket.

This option may be set with `-C` from the command line.

UNIX sockets are filesystem objects that obey UNIX permissions like any other filesystem object. You can set the UNIX sockets' permissions with this option if your webserver would otherwise have no access to the uWSGI socket. When used without a parameter, the permissions will be set to 666. Otherwise the specified chmod value will be used.

chown-socket

Argument: string

Chown UNIX sockets.

umask

Argument: string

Set UNIX socket umask.

freebind

Argument: no argument

Put socket in freebind mode (Linux only).

Allows binding to non-existent network addresses.

map-socket

Argument: string

Map sockets to specific workers.

This option may be declared multiple times.

As you can bind a uWSGI instance to multiple sockets, you can use this option to map specific workers to specific sockets to implement a sort of in-process Quality of Service scheme.

This will map workers 1, 2 and 3 to the first socket and 4 and 5 to the second one.

If you host multiple apps in the same uWSGI instance, you can easily dedicate resources to each of them.

zeromq, zmq, zeromq-socket, zmq-socket

Argument: string

Create a zeromq pub/sub pair.

udp

Argument: string

Run the udp server on the specified address.

Mainly useful for SNMP or shared UDP logging.

See also:

Logging, *The embedded SNMP server*

reuse-port

Argument: no argument

Enable REUSE_PORT flag on socket to allow multiple instances binding on the same address (BSD only).

http-socket-modifier1

Argument: number

Force the specified modifier1 when using HTTP protocol.

http-socket-modifier2

Argument: number

Force the specified modifier2 when using HTTP protocol.

fastcgi-nph-socket

Argument: *add socket*

Bind to the specified UNIX/TCP socket using FastCGI protocol (nph mode).

fastcgi-modifier1

Argument: number

Force the specified modifier1 when using FastCGI protocol.

fastcgi-modifier2

Argument: number

Force the specified modifier2 when using FastCGI protocol.

scgi-socket

Argument: *add socket*

Bind to the specified UNIX/TCP socket using SCGI protocol.

scgi-nph-socket

Argument: *add socket*

Bind to the specified UNIX/TCP socket using SCGI protocol (nph mode).

scgi-modifier1

Argument: number

Force the specified modifier1 when using SCGI protocol.

scgi-modifier2

Argument: number

Force the specified modifier2 when using SCGI protocol.

undeferred-shared-socket

Argument: *add shared socket*

Create a shared socket for advanced jailing or ipc (undeferred mode).

raw-socket

Argument: *add socket no defer*

Bind to the specified UNIX/TCP socket using RAW protocol.

raw-modifier1

Argument: number

Force the specified modifier1 when using RAW protocol.

raw-modifier2

Argument: number

Force the specified modifier2 when using RAW protocol.

puwsgi-socket

Argument: *add socket*

Bind to the specified UNIX/TCP socket using persistent uwsgi protocol (puwsgi).

3.13.2 Process Management

workers, processes

Argument: number

Spawn the specified number of workers/processes.

This option may be set with `-p` from the command line.

Set the number of workers for preforking mode. This is the base for easy and safe concurrency in your app. More workers you add, more concurrent requests you can manage. Each worker corresponds to a system process, so it consumes memory, choose carefully the right number. You can easily drop your system to its knees by setting a too high value. Setting `workers` to a ridiculously high number will *not* magically make your application web scale – quite the contrary.

harakiri

Argument: number

Harakiri timeout in seconds.

Every request that will take longer than the seconds specified in the harakiri timeout will be dropped and the corresponding worker is thereafter recycled.

harakiri-verbose

Argument: no argument

Enable verbose Harakiri mode.

When a request is killed by Harakiri you will get a message in the uWSGI log. Enabling this option will print additional info (for example, the current syscall will be reported on Linux platforms).

harakiri-no-arh, no-harakiri-arh, no-harakiri-after-req-hook

Argument: no argument

Disallow Harakiri killings during after-request hook methods.

mule-harakiri

Argument: number

Set harakiri timeout for mule tasks.

master

Argument: no argument

Enable uWSGI master process.

This option may be set with `-M` from the command line.

reaper

Argument: no argument

Call waitpid(-1,...) after each request to get rid of zombies.

This option may be set with `-r` from the command line.

Enables reaper mode. After each request the server will call `waitpid(-1)` to get rid of zombie processes. If you spawn subprocesses in your app and you happen to end up with zombie processes all over the place you can enable this option. (It really would be better if you could fix your application's process spawning usage though.)

max-requests

Argument: number

Reload workers after the specified amount of managed requests (avoid memory leaks).

This option may be set with `-R` from the command line.

When a worker reaches this number of requests it will get recycled (killed and restarted). You can use this option to "dumb fight" memory leaks. Also take a look at the `reload-on-as` and `reload-on-rss` options as they are more useful for memory leaks.

Beware: The default min-worker-lifetime 60 seconds takes priority over *max-requests*. Do not use with benchmarking as you'll get stalls such as *worker respawning too fast !!! i have to sleep a bit (2 seconds)...*

min-worker-lifetime

Argument: number

A worker cannot be destroyed/reloaded unless it has been alive for N seconds (default 60). This is an anti-fork-bomb measure.

This option is available since version 1.9.

max-worker-lifetime

Argument: number

Reload workers after this many seconds. Disabled by default.

This option is available since version 1.9.

limit-as

Argument: number

Limit process address space (vsz) (in megabytes).

Limits the address space usage of each uWSGI (worker) process using POSIX/UNIX `setrlimit()`. For example, `limit-as 256` will disallow uWSGI processes to grow over 256MB of address space. Address space is the virtual memory a process has access to. It does *not* correspond to physical memory. Read and understand this page before enabling this option: http://en.wikipedia.org/wiki/Virtual_memory

limit-nproc

Argument: number

Limit the number of spawnable processes.

reload-on-as

Argument: number

Reload a worker if its address space usage is higher than the specified value (in megabytes).

reload-on-rss

Argument: number

Reload a worker if its physical unshared memory is higher than the specified value (in megabytes).

evil-reload-on-as

Argument: number

Force the master to reload a worker if its address space is higher than specified megabytes (in megabytes).

evil-reload-on-rss

Argument: number

Force the master to reload a worker if its rss memory is higher than specified megabytes (in megabytes).

threads

Argument: number

Run each worker in prethreaded mode with the specified number of threads per worker.

thread-stacksize, threads-stacksize, thread-stack-size, threads-stack-size

Argument: number

Set threads stacksize.

check-interval

Argument: number **Default:** 1

Set the interval (in seconds) of master checks.

The master process makes a scan of subprocesses, etc. every N seconds. You can increase this time if you need to, but it's DISCOURAGED.

3.13.3 Process Management - Emperor

See also:

The uWSGI Emperor – multi-app deployment

emperor

Argument: string

Run as the Emperor, using the given configuration method.

This option may be declared multiple times.

emperor-freq

Argument: number **Default:** 3

Set the Emperor scanning frequency in seconds.

emperor-pidfile

Argument: string

Write the Emperor pid in the specified file.

emperor-tyrant

Argument: no argument

Put the Emperor in Tyrant (multi-user hosting) mode.

emperor-stats, emperor-stats-server

Argument: string

Run the imperial bureau of statistics on the given address:port.

early-emperor

Argument: no argument

Spawn the emperor before jailing and privilege dropping.

emperor-broodlord

Argument: number

Run the emperor in Broodlord mode.

See also:

Auto-scaling with Broodlord mode

emperor-throttle

Argument: number **Default:** 1000

Set throttling level (in milliseconds) for bad behaving vassals.

emperor-max-throttle

Argument: number **Default:** 180000

Set max throttling level (in milliseconds) for badly behaving vassals (default 3 minutes).

emperor-magic-exec

Argument: no argument

Prefix vassals config files with exec as s:// if they have the executable bit.

imperial-monitor-list, imperial-monitors-list

Argument: no argument

List enabled imperial monitors.

vassals-inherit

Argument: string

Add given config templates to vassals' config.

This works by passing the `inherit` option when starting each vassal (which differs from the `include` option in that `inherit` *will not* replace placeholders etc.).

This option may be declared multiple times.

vassals-include

Argument: string

Add given config templates to vassals' config.

This works by passing the `include` option when starting each vassal (which differs from the `inherit` option in that `include` *will* replace placeholders etc.).

This option may be declared multiple times.

vassals-start-hook

Argument: string

Run the specified command before each vassal starts.

vassals-stop-hook

Argument: string

Run the specified command after vassal's death.

vassal-sos-backlog

Argument: number

Ask emperor for sos if backlog queue has more items than the value specified.

heartbeat

Argument: number

(Vassal option) Announce vassal health to the emperor every N seconds.

emperor-required-heartbeat

Argument: number **Default:** 30

Set the Emperor tolerance about heartbeats.

When a vassal asks for 'heartbeat mode' the emperor will also expect a 'heartbeat' at least every <secs> seconds.

auto-snapshot

Argument: optional number

Automatically make workers snapshot after reload.

See also:

`Snapshot`

reload-mercy

Argument: number

Set the maximum time (in seconds) a worker can take to reload/shutdown.

For example `reload-mercy 8` would brutally kill every worker that will not terminate itself within 8 seconds during graceful reload

3.13.4 Process Management - Zerg

See also:

Zerg mode

zerg

Argument: string

Attach to a zerg server.

This option may be declared multiple times.

zerg-fallback

Argument: no argument

Fallback to normal sockets if the zerg server is not available.

zerg-server

Argument: string

Enable the zerg server on the specified UNIX socket.

zergpool, zerg-pool

Argument: string

Start a zergpool on specified address for specified address (zergpool plugin).

This option may be declared multiple times.

3.13.5 Debugging

backtrace-depth

Argument: number

Set backtrace depth.

memory-report

Argument: no argument

Enable memory usage report.

This option may be set with `-m` from the command line.

When enabled, diagnostic information about RSS and address space usage will be printed in the request log.

profiler

Argument: string

Enable the specified profiler.

dump-options

Argument: no argument

Dump the full list of available options.

show-config

Argument: no argument

Show the current config reformatted as ini.

print

Argument: string

Simple print (for your convenience).

cflags

Argument: no argument

Report uWSGI CFLAGS (useful for building external plugins).

version

Argument: no argument

Print uWSGI version.

allowed-modifiers

Argument: string

Comma separated list of allowed modifiers for clients.

`allowed-modifiers 0,111` would allow access to only the WSGI handler and the cache handler.

connect-and-read

Argument: *str*

Connect to a socket and wait for data from it.

3.13.6 Configuration

See also:

Configuring uWSGI

strict

Argument: no argument

Enable strict configuration parsing. If any unknown option is encountered in a configuration file, an error is shown and uWSGI quits.

To use placeholder variables when using strict mode, use the `set-placeholder` option.

set

Argument: string

Set a configuration option. This option was created to work around a specific problem with commandline options on Solaris and should not normally need to be used.

set-placeholder

Argument: string

Set a placeholder variable. The value of this option should be in the form `placeholder=value`. This option can be to set placeholders when the `strict` option is enabled.

This option is available since version 1.9.18.

declare-option

Argument: string

Declare a new custom uWSGI option.

inherit

Argument: string

Use the specified file as configuration template. The file type of the included file is automatically detected based on filename extension.

Note that environment variables, external file includes and placeholders are *not* expanded inside the inherited configuration. Magic variables (*e.g.* `%n`) are expanded normally.

See also:

How uWSGI parses config files

include

Argument: string

Include the specified file as if its configuration entries had been declared here. The file type of the included file is automatically detected based on filename extension.

This option is available since version 1.3.

plugins, plugin

Argument: string

Load uWSGI plugins (comma-separated).

This option may be declared multiple times.

plugins-dir, plugin-dir

Argument: string

Add a directory to uWSGI plugin search path.

This option may be declared multiple times.

plugins-list, plugin-list

Argument: no argument

List enabled plugins.

autoload

Argument: no argument

Try to automatically load plugins when unknown options are found.

dlopen

Argument: string

Blindly load a shared library.

ini

Argument: number

Load config from ini file.

xml, xmlconfig

Argument: string

Load XML file as configuration.

This option may be set with `-x` from the command line.

yaml, yal

Argument: string

Load config from yaml file.

This option may be set with `-y` from the command line.

json, js

Argument: string

Load config from json file.

This option may be set with `-j` from the command line.

sqlite3, sqlite

Argument: number

Load config from sqlite3 db.

ldap

Argument: number

Load configuration from ldap server.

See also:

Configuring uWSGI with LDAP

ldap-schema

Argument: no argument

Dump uWSGI ldap schema.

See also:

Configuring uWSGI with LDAP

ldap-schema-ldif

Argument: no argument

Dump uWSGI ldap schema in ldif format.

See also:

Configuring uWSGI with LDAP

3.13.7 Config logic

See also:

Configuration logic

for

Argument: string

For cycle.

endfor

Argument: optional string

End for cycle.

if-opt

Argument: string

Check for option.

if-not-opt

Argument: string

Check for lack of option.

if-env, ifenv

Argument: string

Check for environment variable.

if-not-env

Argument: string

Check for lack of environment variable.

if-reload

Argument: string

Check for reload.

if-not-reload

Argument: string

Check for lack of reload.

if-exists, ifexists

Argument: string

Check for file/directory existence.

if-not-exists

Argument: string

Check for file/directory inexistence.

if-file

Argument: string

Check for file existence.

if-not-file

Argument: string

Check for file inexistence.

if-dir, ifdir, if-directory

Argument: string

Check for directory existence.

if-not-dir

Argument: string

Check for directory inexistence.

endif

Argument: optional string

End if block.

3.13.8 Logging

See also:

Logging

disable-logging

Argument: no argument

Disable request logging.

This option may be set with `-L` from the command line.

When enabled, only uWSGI internal messages and errors are logged.

ignore-sigpipe

Argument: no argument

Do not report (annoying) SIGPIPE.

ignore-write-errors

Argument: no argument

Do not report (annoying) write()/writev() errors.

write-errors-tolerance

Argument: number

Set the maximum number of allowed write errors (default: no tolerance).

write-errors-exception-only

Argument: no argument

Only raise an exception on write errors giving control to the app itself.

disable-write-exception

Argument: no argument

Disable exception generation on write()/writev().

logto

Argument: string

Set logfile/udp address.

logto2

Argument: string

Log to specified file or udp address after privileges drop.

log-format, logformat

Argument: string

Set advanced format for request logging.

logformat-strftime, log-format-strftime

Argument: no argument

Apply strftime to logformat output.

logfile-chown

Argument: no argument

Chown logfiles.

logfile-chmod

Argument: string

Chmod logfiles.

log-syslog

Argument: optional string

Log to syslog.

Passing a parameter makes uwsgi use the parameter as program-name in the syslog entry header.

log-socket

Argument: string

Send logs to the specified socket.

logger

Argument: string

Set/append a logger.

This option may be declared multiple times.

logger-list, loggers-list

Argument: no argument

List enabled loggers.

threaded-logger

Argument: no argument

Offload log writing to a thread.

log-drain

Argument: *regexp*

Drain (do not show) log lines matching the specified regexp.

This option may be declared multiple times.

log-zeromq

Argument: string

Send logs to a ZeroMQ server.

log-master

Argument: no argument

Delegate logging to master process.

Delegate the write of the logs to the master process (this will put all of the logging I/O to a single process). Useful for system with advanced I/O schedulers/elevators.

log-master-bufsize

Argument: number

Set the buffer size for the master logger. Log messages larger than this will be truncated.

log-reopen

Argument: no argument

Reopen log after reload.

log-truncate

Argument: no argument

Truncate log on startup.

log-maxsize

Argument: number

Set maximum logfile size.

log-backupname

Argument: string

Set logfile name after rotation.

log-prefix, logdate, log-date

Argument: optional string

Prefix log lines with date (without argument) or a strftime string.

log-zero

Argument: no argument

Log responses without body (zero response size).

log-slow

Argument: number

Log requests slower than the specified number of milliseconds.

log-4xx

Argument: no argument

Log requests with a 4xx response.

log-5xx

Argument: no argument

Log requests with a 5xx response.

log-big

Argument: number

Log requestes bigger than the specified size in bytes.

log-sendfile

Argument: no argument

Log sendfile requests.

log-micros

Argument: no argument

Report response time in microseconds instead of milliseconds.

log-x-forwarded-for

Argument: no argument

Use the ip from X-Forwarded-For header instead of REMOTE_ADDR. Used when uWSGI is run behind multiple proxies.

stats, stats-server

Argument: string

Enable the stats server on the specified address.

ssl-verbose

Argument: no argument

Be verbose about SSL errors.

snmp

Argument: string

Enable the embedded SNMP server.

This option may be declared multiple times.

See also:

The embedded SNMP server

snmp-community

Argument: string

Set the SNMP community string.

3.13.9 Alarms

See also:

The uWSGI alarm subsystem (from 1.3)

alarm

Argument: string

Create a new alarm. Syntax: <alarm> <plugin:args>.

This option may be declared multiple times.

alarm-freq

Argument: number

Tune the alarm anti-loop system (default 3 seconds).

log-alarm

Argument: string

Raise the specified alarm when a log line matches the specified regexp, syntax: <alarm>[,alarm...] <regexp>.

This option may be declared multiple times.

alarm-list, alarms-list

Argument: no argument

List enabled alarms.

3.13.10 uWSGI Process

daemonize

Argument: *logfile*

Daemonize uWSGI and write messages into given log file or UDP socket address.

See also:

Logging

daemonize2

Argument: *logfile*

Daemonize uWSGI after loading application, write messages into given log file or UDP socket address.

See also:

Logging

stop

Argument: *pidfile*

Send the stop (SIGINT) signal to the instance described by the pidfile.

See also:

Managing the uWSGI server

reload

Argument: *pidfile*

Send the reload (SIGHUP) signal to the instance described by the pidfile.

See also:

Managing the uWSGI server

pause

Argument: *pidfile*

Send the pause (SIGTSTP) signal to the instance described by the pidfile.

See also:

Managing the uWSGI server

suspend

Argument: *pidfile*

Send the suspend (SIGTSTP) signal to the instance described by the pidfile.

See also:

Managing the uWSGI server

resume

Argument: *pidfile*

Send the resume (SIGTSTP) signal to the instance described by the pidfile.

See also:

Managing the uWSGI server

auto-procname

Argument: no argument

Automatically set process name to something meaningful.

Generated process names may be 'uWSGI Master', 'uWSGI Worker #', etc.

procname-prefix

Argument: string

Add prefix to process names.

procname-prefix-spaced

Argument: string

Add spaced prefix to process names.

procname-append

Argument: string

Append string to process names.

procname

Argument: string

Set process name to given value.

procname-master

Argument: string

Set master process name to given value.

pidfile

Argument: string

Create pidfile (before privileges drop).

pidfile2

Argument: string

Create pidfile (after privileges drop).

chroot

Argument: string

Chroot() to the specified directory.

uid

Argument: *username|uid*

Setuid to the specified user/uid.

gid

Argument: *groupname|gid*

Setgid to the specified grooup/gid.

no-initgroups

Argument: no argument

Disable additional groups set via initgroups().

cap

Argument: string

Set process capability.

unshare

Argument: string

Unshare() part of the processes and put it in a new namespace.

exec-pre-jail

Argument: string

Run the specified command before jailing.

This option may be declared multiple times.

exec-post-jail

Argument: string

Run the specified command after jailing.

This option may be declared multiple times.

exec-in-jail

Argument: string

Run the specified command in jail after initialization.

This option may be declared multiple times.

exec-as-root

Argument: string

Run the specified command before privileges drop.

This option may be declared multiple times.

exec-as-user

Argument: string

Run the specified command after privileges drop.

This option may be declared multiple times.

exec-as-user-atexit

Argument: string

Run the specified command before app exit and reload.

This option may be declared multiple times.

exec-pre-app

Argument: string

Run the specified command before app loading.

This option may be declared multiple times.

cgroup

Argument: string

Put the processes in the specified cgroup (Linux only).

This option may be declared multiple times.

See also:

Running uWSGI in a Linux CGroup

cgroup-opt

Argument: string

Set value in specified cgroup option.

This option may be declared multiple times.

See also:

Running uWSGI in a Linux CGroup

namespace, ns

Argument: string

Run in a new namespace under the specified rootfs.

See also:

Jailing your apps using Linux Namespaces

namespace-keep-mount

Argument: *<mount>[:<jailed-mountpoint>]*

Keep the specified mountpoint in your namespace, optionally renaming it.

See also:

Jailing your apps using Linux Namespaces

namespace-net, ns-net

Argument: string

Add network namespace.

forkbomb-delay

Argument: number

Sleep for the specified number of seconds when a forkbomb is detected.

binary-path

Argument: string

Force binary path.

If you do not have uWSGI in the system path you can force its path with this option to permit the reloading system and the Emperor to easily find the binary to execute.

privileged-binary-patch

Argument: string

Patch the uwsgi binary with a new command (before privileges drop).

unprivileged-binary-patch

Argument: string

Patch the uwsgi binary with a new command (after privileges drop).

privileged-binary-patch-arg

Argument: string

Patch the uwsgi binary with a new command and arguments (before privileges drop).

unprivileged-binary-patch-arg

Argument: string

Patch the uwsgi binary with a new command and arguments (after privileges drop).

async

Argument: number

Enable async mode with specified cores.

See also:

uWSGI asynchronous/non-blocking modes (updated to uWSGI 1.9)

max-fd

Argument: number

Set maximum number of file descriptors (requires root privileges).

master-as-root

Argument: no argument

Leave master process running as root.

3.13.11 Miscellaneous

thunder-lock

Argument: no argument

Serialize accept() usage (if possibie).

skip-zero

Argument: no argument

Skip check of file descriptor 0.

need-app

Argument: no argument

Exit if no app can be loaded.

exit-on-reload

Argument: no argument

Force exit even if a reload is requested.

die-on-term

Argument: no argument

Exit instead of brutal reload on SIGTERM.

no-fd-passing

Argument: no argument

Disable file descriptor passing.

single-interpreter

Argument: no argument

Do not use multiple interpreters (where available).

This option may be set with `-i` from the command line.

Some of the supported languages (such as Python) have the concept of "multiple interpreters". This feature allows you to isolate apps living in the same process. If you do not want this kind of feature use this option.

max-apps

Argument: number

Set the maximum number of per-worker applications.

sharedarea

Argument: number

Create a raw shared memory area of specified number of pages.

This option may be set with `-A` from the command line.

This enables the SharedArea. This is a low level shared memory. If you want a more usable/user-friendly system look at the caching framework.

See also:

SharedArea – share memory pages between uWSGI components

cgi-mode

Argument: no argument

Force CGI-mode for plugins supporting it.

This option may be set with `-c` from the command line.

When enabled, responses generated by uWSGI will not be HTTP responses, but CGI ones; namely, the `Status:` header will be added.

buffer-size

Argument: number **Default:** 4096

Set the internal buffer size for uwsgi packet parsing.

This option may be set with `-b` from the command line.

If you plan to receive big requests with lots of headers you can increase this value up to 64k (65535).

enable-threads

Argument: no argument

Enable threads.

This option may be set with `-T` from the command line.

Enable threads in the embedded languages. This will allow to spawn threads in your app.

> **Warning:** Threads will simply *not work* if this option is not enabled. There will likely be no error, just no execution of your thread code.

signal-bufsize, signals-bufsize

Argument: number

Set buffer size for signal queue.

socket-timeout

Argument: number **Default:** 4

Set internal sockets timeout in seconds.

This option may be set with `-z` from the command line.

max-vars

Argument: number

Set the amount of internal iovec/vars structures for uwsgi clients (web servers, etc.).

This option may be set with `-v` from the command line.

This is only a security measure you will probably never need to touch.

weight

Argument: number

Weight of the instance (used by clustering/lb/subscriptions).

auto-weight

Argument: number

Set weight of the instance (used by clustering/lb/subscriptions) automatically.

no-server

Argument: no argument

Initialize the uWSGI server but exit as soon as the initialization is complete (useful for testing).

command-mode

Argument: no argument

Force command mode.

no-defer-accept

Argument: no argument

Disable deferred `accept()` on sockets.

by default (where available) uWSGI will defer the accept() of requests until some data is sent by the client (this is a security/performance measure). If you want to disable this feature for some reason, specify this option.

so-keepalive

Argument: no argument

Enable TCP KEEPALIVEs.

never-swap

Argument: no argument

Lock all memory pages avoiding swapping.

ksm

Argument: number

Enable Linux KSM.

This option may be declared multiple times.

touch-reload

Argument: string

Reload uWSGI if the specified file or directory is modified/touched.

This option may be declared multiple times.

touch-logrotate

Argument: string

Trigger logrotation if the specified file is modified/touched.

This option may be declared multiple times.

touch-logreopen

Argument: string

Trigger log reopen if the specified file is modified/touched.

This option may be declared multiple times.

propagate-touch

Argument: no argument

Over-engineering option for system with flaky signal mamagement.

no-orphans

Argument: no argument

Automatically kill workers if master dies (can be dangerous for availability).

prio

Argument: number

Set processes/threads priority (`nice`) value.

cpu-affinity

Argument: *number of cores for each worker (Linux only)*

Set CPU affinity.

Set the number of cores (CPUs) to allocate to each worker process.

For example

- With 4 workers, 4 CPUs and `cpu-affinity` is 1, each worker is allocated one CPU.
- With 4 workers, 2 CPUs and `cpu-affinity` is 1, workers get one CPU each (0; 1; 0; 1).
- With 4 workers, 4 CPUs and `cpu-affinity` is 2, workers get two CPUs each in a round-robin fashion (0, 1; 2, 3; 0, 1; 2; 3).
- With 8 workers, 4 CPUs and `cpu-affinity` is 3, workers get three CPUs each in a round-robin fashion (0, 1, 2; 3, 0, 1; 2, 3, 0; 1, 2, 3; 0, 1, 2; 3, 0, 1; 2, 3, 0; 1, 2, 3).

remap-modifier

Argument: string

Remap request modifier from one id to another (old-id:new-id).

env

Argument: string

Set environment variable (key=value).

unenv

Argument: string

Set environment variable (key).

close-on-exec

Argument: no argument

Set close-on-exec on sockets (could be required for spawning processes in requests).

mode

Argument: string

Set uWSGI custom mode.

Generic *mode* option that is passed down to applications as `uwsgi.mode` (or similar for other languages)

vacuum

Argument: no argument

Try to remove all of the generated files/sockets (UNIX sockets and pidfiles) upon exit.

cron

Argument: string

Add a cron task.

worker-exec

Argument: string

Run the specified command as worker instead of uWSGI itself.

This could be used to run a PHP FastCGI server pool:

```
/usr/bin/uwsgi --workers 4 --worker-exec /usr/bin/php53-cgi
```

attach-daemon

Argument: string

Attach a command/daemon to the master process (the command has to remain in foreground).

This will allow the uWSGI master to control/monitor/respawn this process.

A typical usage is attaching a `memcached` instance:

```
[uwsgi]
master = true
attach-daemon = memcached
```

smart-attach-daemon

Argument: *pidfile*

Attach a command/daemon to the master process managed by a pidfile (the command must daemonize).

smart-attach-daemon2

Argument: *pidfile*

Attach a command/daemon to the master process managed by a pidfile (the command must NOT daemonize).

3.13.12 Locks

See also:

Locks

locks

Argument: number

Create the specified number of shared locks.

lock-engine

Argument: string

Set the lock engine.

ftok

Argument: string

Set the ipcsem key via ftok() for avoiding duplicates.

flock

Argument: string

Lock the specified file before starting, exit if locked.

flock-wait

Argument: string

Lock the specified file before starting, wait if locked.

flock2

Argument: string

Lock the specified file after logging/daemon setup, exit if locked.

flock-wait2

Argument: string

Lock the specified file after logging/daemon setup, wait if locked.

3.13.13 Cache

See also:

The uWSGI caching framework

cache

Argument: number

Create a shared cache containing given elements.

cache-blocksize

Argument: number **Default:** 65536

Set the cache block size in bytes. It's a good idea to use a multiple of 4096 (common memory page size).

cache-store

Argument: string

Enable persistent cache to disk.

cache-store-sync

Argument: number

Set frequency of sync for persistent cache.

cache-server

Argument: string

Enable the threaded cache server.

cache-server-threads

Argument: number

Set the number of threads for the cache server.

cache-no-expire

Argument: no argument

Disable auto sweep of expired items.

cache-expire-freq

Argument: number

Set the frequency of cache sweeper scans (default 3 seconds).

cache-report-freed-items

Argument: no argument

Constantly report the cache item freed by the sweeper (use only for debug).

3.13.14 Queue

See also:

The uWSGI queue framework

queue

Argument: number

Enable the shared queue with the given size.

queue-blocksize

Argument: number

Set the block size for the queue.

queue-store

Argument: *filename*

Enable periodical persisting of the queue to disk.

queue-store-sync

Argument: number

Set periodical persisting frequency in seconds.

3.13.15 Spooler

See also:

The uWSGI Spooler

spooler

Argument: string

Run a spooler on the specified directory.

This option may be set with `-Q` from the command line.

spooler-external

Argument: string

Map spooler requests to a spooler directory, but do not start a spooler (spooling managed by external instance).

spooler-ordered

Argument: no argument

Try to order the execution of spooler tasks.

spooler-chdir

Argument: string

Chdir() to specified directory before each spooler task.

spooler-processes

Argument: number

Set the number of processes for spoolers.

spooler-quiet

Argument: no argument

Do not be verbose with spooler tasks.

spooler-max-tasks

Argument: number

Set the maximum number of tasks to run before recycling a spooler.

spooler-harakiri

Argument: number

Set harakiri timeout for spooler tasks.

3.13.16 Mules

See also:

uWSGI Mules

mule

Argument: string

Add a mule (signal-only mode without argument).

This option may be declared multiple times.

mules

Argument: number

Add the specified number of mules.

farm

Argument: string

Add a mule farm.

signal

Argument: string

Send a uwsgi signal to a server.

3.13.17 Application loading

chdir

Argument: string

Chdir to specified directory before apps loading.

chdir2

Argument: string

Chdir to specified directory after apps loading.

lazy

Argument: no argument

Set lazy mode (load apps in workers instead of master).

This option may have memory usage implications as Copy-on-Write semantics can not be used. When `lazy` is enabled, only workers will be reloaded by uWSGI's reload signals; the master will remain alive. As such, uWSGI configuration changes are not picked up on reload by the master.

lazy-apps

Argument: no argument

Load apps in each worker instead of the master.

This option may have memory usage implications as Copy-on-Write semantics can not be used. Unlike `lazy`, this only affects the way applications are loaded, not master's behavior on reload.

cheap

Argument: no argument

Set cheap mode (spawn workers only after the first request).

cheaper

Argument: number

Set cheaper mode (adaptive process spawning).

This an advanced *cheap* mode. This will only spawn <n> workers on startup and will use various (pluggable) algorithms to implement adaptive process spawning.

cheaper-initial

Argument: number

Set the initial number of processes to spawn in cheaper mode.

cheaper-algo

Argument: string

Choose to algorithm used for adaptive process spawning).

cheaper-step

Argument: number

Number of additional processes to spawn at each overload.

cheaper-overload

Argument: number

Increase workers after specified overload.

cheaper-algo-list, cheaper-algos-list, cheaper-list

Argument: no argument

List enabled 'cheaper' algorithms.

idle

Argument: number

Set idle mode (put uWSGI in cheap mode after inactivity).

die-on-idle

Argument: no argument

Shutdown uWSGI when idle.

mount

Argument: */mountpoint=/app/path*

Load application under mountpoint.

Example: `mount /pinax=/var/www/pinax/deploy/pinax.wsgi`

worker-mount

Argument: string

Load application under mountpoint in the specified worker or after workers spawn.

This option may be declared multiple times.

grunt

Argument: no argument

Enable grunt mode (in-request fork).

3.13.18 Request handling

limit-post

Argument: number

Limit request body (bytes) based on the `CONTENT_LENGTH` uwsgi var.

post-buffering

Argument: number

Enable post buffering past N bytes.

Enables HTTP body buffering. uWSGI will save to disk all HTTP bodies larger than the limit specified. This option is required and auto-enabled for Ruby Rack applications as they require a rewindable input stream.

post-buffering-bufsize

Argument: number

Set buffer size for read() in post buffering mode.

This is an advanced option you probably won't need to touch.

upload-progress

Argument: string

Enable creation of .json files in the specified directory during a file upload.

Enable the embedded upload progress system.

Pass the name of a directory where uWSGI has write permissions into.

For every upload with a `X-Progress-ID` query string ("GET") parameter, a JSON file will be written to this directory containing the status of the upload. AJAX calls can then be used to read these files.

For instance, when `upload-progress` is set to `/var/www/progress` the user uploads a file to the URL:

```
/upload?X-Progress-ID=550e8400-e29b-41d4-a716-446655440000
```

uWSGI find `X-Progress-ID` in the query string and create a file called `/var/www/progress/550e8400-e29b-41d4-a716-446655440000.js` containing something like:

```
{"state": "uploading", "received": 170000, "size": 300000}
```

If `/var/www/progress` has been mapped to the `/progress` path in your web server, you can then request this file at `/progress/550e8400-e29b-41d4-a716-446655440000.js`.

It's likely that your web server supports similar functionality (Nginx does, at least), but the uWSGI implementation is ostensibly more controllable and hackable.

no-default-app

Argument: no argument

Do not fallback to default app.

By default, when uWSGI does not find a corresponding app for the specified `SCRIPT_NAME` variable, it will use the default app (most of the time the app mounted under /). Enabling this option will return an error in case of unavailable app.

manage-script-name

Argument: no argument

Automatically rewrite SCRIPT_NAME and PATH_INFO.

If for some reason your webserver cannot manage `SCRIPT_NAME` on its own you can force uWSGI to rebuild the `PATH_INFO` variable automatically from it.

ignore-script-name

Argument: no argument

Ignore SCRIPT_NAME.

catch-exceptions

Argument: no argument

Report exception as HTTP output.

> **Warning:** This option is heavily discouraged as it is a definite security risk.

reload-on-exception

Argument: no argument

Reload a worker when an exception is raised.

reload-on-exception-type

Argument: string

Reload a worker when a specific exception type is raised.

This option may be declared multiple times.

reload-on-exception-value

Argument: string

Reload a worker when a specific exception value is raised.

This option may be declared multiple times.

reload-on-exception-repr

Argument: string

Reload a worker when a specific exception type+value (language-specific) is raised.

This option may be declared multiple times.

add-header

Argument: string

Automatically add HTTP headers to response.

This option may be declared multiple times.

vhost

Argument: no argument

Enable virtualhosting mode (based on SERVER_NAME variable).

See also:

`VirtualHosting`

vhost-host

Argument: no argument

Enable virtualhosting mode (based on HTTP_HOST variable).

By default the virtualhosting mode use the SERVER_NAME variable as the hostname key. If you want to use the HTTP_HOST one (corresponding to the Host: header) add this option

See also:

`VirtualHosting`

3.13.19 Clustering

multicast

Argument: string

Subscribe to specified multicast group. internal option, usable by third party plugins.

multicast-ttl

Argument: number

Set multicast ttl.

cluster

Argument: string

Join specified uWSGI cluster.

cluster-nodes

Argument: *address:port*

Get nodes list from the specified cluster without joining it.

This list is used internally by the uwsgi load balancing api.

cluster-reload

Argument: *address:port*

Send a graceful reload message to the cluster.

cluster-log

Argument: *address:port*

Send a log line to the cluster.

For instance, `--cluster-log "Hello, world!"` will print that to each cluster node's log file.

3.13.20 Subscriptions

See also:

uWSGI Subscription Server

subscriptions-sign-check

Argument: string

Set digest algorithm and certificate directory for secured subscription system.

subscriptions-sign-check-tolerance

Argument: number

Set the maximum tolerance (in seconds) of clock skew for secured subscription system.

subscription-algo

Argument: string

Set load balancing algorithm for the subscription system.

subscription-dotsplit

Argument: no argument

Try to fallback to the next part (dot based) in subscription key.

subscribe-to, st, subscribe

Argument: string

Subscribe to the specified subscription server.

This option may be declared multiple times.

subscribe-freq

Argument: number

Send subscription announce at the specified interval.

subscription-tolerance

Argument: number

Set tolerance for subscription servers.

unsubscribe-on-graceful-reload

Argument: no argument

Force unsubscribe request even during graceful reload.

3.13.21 Router

See also:

uWSGI internal routing

route

Argument: string

Add a route.

This option may be declared multiple times.

route-host

Argument: string

Add a route based on Host header.

This option may be declared multiple times.

route-uri

Argument: string

Add a route based on REQUEST_URI.

This option may be declared multiple times.

route-qs

Argument: string

Add a route based on QUERY_STRING.

This option may be declared multiple times.

router-list, routers-list

Argument: no argument

List enabled routers.

3.13.22 Static files

static-check, check-static

Argument: string

Check for static files in the specified directory.

This option may be declared multiple times.

Specify a directory that uWSGI will check before passing control to a specific handler.

uWSGI will check if the requested `PATH_INFO` has a file correspondence in this directory and serve it.

For example, with `check-static /var/www/example.com`, uWSGI will check if `/var/www/example.com/foo.png` exists and directly serve it using *sendfile()* (or another configured method).

check-static-docroot

Argument: no argument

Check for static files in the requested DOCUMENT_ROOT.

static-map

Argument: string

Map mountpoint to static directory (or file).

This option may be declared multiple times.

Whenever a PATH_INFO starts with one of the configured resources, uWSGI will serve the file as a static file.

static-map2

Argument: string

Map mountpoint to static directory (or file), completely appending the requested resource to the docroot.

This option may be declared multiple times.

static-skip-ext

Argument: string

Skip specified extension from staticfile checks.

This option may be declared multiple times.

static-index

Argument: string

Search for specified file if a directory is requested.

This option may be declared multiple times.

With `static-index=index.html`, if the client asks for `/doc/` then uWSGI will check for `/doc/index.html` and if it exists it will be served to the client.

mimefile, mime-file

Argument: string

Set mime types file path (default /etc/mime.types).

This option may be declared multiple times.

static-expires-type

Argument: string

Set the Expires header based on content type (syntax: Content-type=Expires).

This option may be declared multiple times.

static-expires-type-mtime

Argument: string

Set the Expires header based on content type and file mtime (syntax: Content-type=Expires).

This option may be declared multiple times.

static-expires

Argument: string

Set the Expires header based on filename regexp (syntax x=y).

This option may be declared multiple times.

static-expires-mtime

Argument: string

Set the Expires header based on filename regexp and file mtime (syntax x=y).

This option may be declared multiple times.

static-expires-uri

Argument: string

Set the Expires header based on REQUEST_URI regexp (syntax x=y).

This option may be declared multiple times.

static-expires-uri-mtime

Argument: string

Set the Expires header based on REQUEST_URI regexp and file mtime (syntax x=y).

This option may be declared multiple times.

static-expires-path-info

Argument: string

Set the Expires header based on PATH_INFO regexp (syntax x=y).

This option may be declared multiple times.

static-expires-path-info-mtime

Argument: string

Set the Expires header based on PATH_INFO regexp and file mtime (syntax x=y).

This option may be declared multiple times.

static-offload-to-thread

Argument: number

Offload static file serving to a thread (upto the specified number of threads).

file-serve-mode

Argument: string

Set static file serving mode (x-sendfile, nginx, ...).

Set the static serving mode:

- `x-sendfile` will use the X-Sendfile header supported by Apache, Cherokee, lighttpd
- `x-accel-redirect` will use the X-Accel-Redirect header supported by Nginx

By default the *sendfile()* syscall is used.

check-cache

Argument: no argument

Check for response data in the cache based on PATH_INFO.

3.13.23 Clocks

clock

Argument: string

Set a clock source.

clock-list, clocks-list

Argument: no argument

List enabled clocks.

3.13.24 Loop engines

loop

Argument: string

Select the uWSGI loop engine (advanced).

See also:

`LoopEngine`

loop-list, loops-list

Argument: no argument

List enabled loop engines.

3.13.25 Greenlet

See also:

Greenlet

greenlet

Argument: no argument

Enable greenlet as suspend engine.

3.13.26 Gevent

See also:

The Gevent loop engine

gevent

Argument: number

A shortcut enabling gevent loop engine with the specified number of async cores and optimal parameters.

gevent-monkey-patch

Argument: no argument

Call gevent.monkey.patch_all() automatically on startup.

gevent-wait-for-hub

Argument: no argument

Wait for gevent hub's death instead of the control greenlet.

3.13.27 Stackless

See also:

Stackless

stackless

Argument: no argument

Use stackless as suspend engine.

3.13.28 uGreen

See also:

uGreen – uWSGI Green Threads

ugreen

Argument: no argument

Enable uGreen as suspend/resume engine.

ugreen-stacksize

Argument: number

Set ugreen stack size in pages.

3.13.29 Fiber

fiber

Argument: no argument

Enable Ruby fiber as suspend engine.

3.13.30 CoroAE

coroae

Argument: *setup coroae*

A shortcut enabling Coro::AnyEvent loop engine with the specified number of async cores and optimal parameters.

3.13.31 tornado

tornado

Argument: *setup tornado*

A shortcut enabling tornado loop engine with the specified number of async cores and optimal parameters.

3.13.32 Carbon

See also:

Integration with Graphite/Carbon

carbon

Argument: *host:port*

Push statistics to the specified carbon server/port.

This option may be declared multiple times.

carbon-timeout

Argument: number **Default:** 3

Set Carbon connection timeout in seconds.

carbon-freq

Argument: number **Default:** 60

Set Carbon push frequency in seconds.

carbon-id

Argument: string

Set the identifier for Carbon metrics (by default the first uWSGI socket name).

carbon-no-workers

Argument: no argument

Disable generation of single worker metrics.

carbon-max-retry

Argument: number

Set maximum number of retries in case of connection errors (default 1).

carbon-retry-delay

Argument: number

Set connection retry delay in seconds (default 7).

carbon-root

Argument: string

Set carbon metrics root node (default 'uwsgi').

carbon-hostname-dots

Argument: string

Set char to use as a replacement for dots in hostname (dots are not replaced by default).

carbon-name-resolve, carbon-resolve-names

Argument: no argument

Allow using hostname as carbon server address (default disabled).

carbon-idle-avg

Argument: string

Average values source during idle period (no requests), can be 'last', 'zero', 'none' (default is last).

carbon-use-metrics

Argument: no argument

Don't compute all statistics, use metrics subsystem data instead (warning! key names will be different).

3.13.33 CGI

See also:

Running CGI scripts on uWSGI

cgi

Argument: *[mountpoint=]script*

Add a CGI directory/script with optional mountpoint (URI prefix).

cgi-map-helper, cgi-helper

Argument: *extension=helper-executable*

Add a cgi helper to map an extension into an executable.

cgi-from-docroot

Argument: no argument

Blindly enable cgi in DOCUMENT_ROOT.

cgi-buffer-size

Argument: number

Set the CGI buffer size.

cgi-timeout

Argument: number

Set CGI script timeout.

cgi-index

Argument: string

Add a CGI index file.

This option may be declared multiple times.

cgi-allowed-ext

Argument: string

Allowed extensions for CGI.

This option may be declared multiple times.

cgi-unset

Argument: string

Unset specified environment variables before running CGI executable.

This option may be declared multiple times.

cgi-loadlib

Argument: string

Load a CGI shared library/optimizer.

This option may be declared multiple times.

cgi-optimize, cgi-optimized

Argument: no argument

Enable CGI realpath() optimizer.

cgi-path-info

Argument: no argument

Disable PATH_INFO management in CGI scripts.

3.13.34 Busyness Cheaper algorithm

See also:

The uWSGI cheaper subsystem – adaptive process spawning

cheaper-busyness-max

Argument: number

Set the cheaper busyness high percent limit, above that value worker is considered loaded (default 50).

cheaper-busyness-min

Argument: number

Set the cheaper busyness low percent limit, belowe that value worker is considered idle (default 25).

cheaper-busyness-multiplier

Argument: number

Set initial cheaper multiplier, worker needs to be idle for cheaper-overload*multiplier seconds to be cheaped (default 10).

cheaper-busyness-penalty

Argument: number

Penalty for respawning workers to fast, it will be added to the current multiplier value if worker is cheaped and than respawned back too fast (default 2).

cheaper-busyness-verbose

Argument: no argument

Enable verbose log messages from busyness algorithm.

cheaper-busyness-backlog-alert

Argument: number

Spawn emergency worker if anytime listen queue is higher than this value (default 33) (Linux only).

cheaper-busyness-backlog-multiplier

Argument: number

Set cheaper multiplier used for emergency workers (default 3) (Linux only).

cheaper-busyness-backlog-step

Argument: number

Number of emergency workers to spawn at a time (default 1) (Linux only).

cheaper-busyness-backlog-nonzero

Argument: number

Spawn emergency worker(s) if backlog is > 0 for more then N seconds (default 60).

3.13.35 Erlang

See also:

Integrating uWSGI with Erlang

erlang

Argument: string

Spawn an Erlang c-node.

erlang-cookie

Argument: string

Set Erlang cookie.

3.13.36 Fastrouter

See also:

The uWSGI FastRouter

fastrouter

Argument: *address:port*

Run the fastrouter (uwsgi protocol proxy/load balancer) on the specified address:port.

fastrouter-processes, fastrouter-workers

Argument: number

Prefork the specified number of fastrouter processes.

fastrouter-zerg

Argument: *corerouter zerg*

Attach the fastrouter to a zerg server.

fastrouter-use-cache

Argument: no argument

Use uWSGI cache as hostname->server mapper for the fastrouter.

fastrouter-use-pattern

Argument: *corerouter use pattern*

Use a pattern for fastrouter hostname->server mapping.

fastrouter-use-base

Argument: *corerouter use base*

Use a base dir for fastrouter hostname->server mapping.

fastrouter-fallback

Argument: string

Fallback to the specified node in case of error.

This option may be declared multiple times.

fastrouter-use-cluster

Argument: no argument

Load balance to nodes subscribed to the cluster.

fastrouter-use-code-string

Argument: *corerouter cs*

Use code string as hostname->server mapper for the fastrouter.

fastrouter-use-socket

Argument: optional *corerouter use socket*

Forward request to the specified uwsgi socket.

fastrouter-to

Argument: string

Forward requests to the specified uwsgi server (you can specify it multiple times for load balancing).

This option may be declared multiple times.

fastrouter-gracetime

Argument: number

Retry connections to dead static nodes after the specified amount of seconds.

fastrouter-events

Argument: number

Set the maximum number of concurrent events the fastrouter can return in one cycle.

fastrouter-quiet

Argument: no argument

Do not report failed connections to instances.

fastrouter-cheap

Argument: no argument

Run the fastrouter in cheap mode (do not respond to requests unless a node is available).

fastrouter-subscription-slot

Argument: *deprecated*

*** deprecated ***.

fastrouter-subscription-server

Argument: *corerouter ss*

Add a Subscription Server to the fastrouter to build the hostname:address map.

See also:

uWSGI Subscription Server

fastrouter-timeout

Argument: number

Set the internal fastrouter timeout.

fastrouter-post-buffering

Argument: number

Enable fastrouter post buffering.

fastrouter-post-buffering-dir

Argument: string

Put fastrouter buffered files to the specified directory.

fastrouter-stats, fastrouter-stats-server, fastrouter-ss

Argument: string

Run the fastrouter stats server.

fastrouter-harakiri

Argument: number

Enable fastrouter harakiri.

3.13.37 GeoIP

geoip-country

Argument: string

Load the specified geoip country database.

geoip-city

Argument: string

Load the specified geoip city database.

3.13.38 GlusterFS

glusterfs-mount

Argument: string

Virtual mount the specified glusterfs volume in a uri.

This option may be declared multiple times.

glusterfs-timeout

Argument: number

Timeout for glusterfs async mode.

3.13.39 Gccgo

See also:

uWSGI Go support (1.4 only)

go-load, gccgo-load

Argument: string

Load a go shared library in the process address space, eventually patching main.main and __go_init_main.

This option may be declared multiple times.

go-args, gccgo-args

Argument: string

Set go commandline arguments.

3.13.40 Go_plugin

See also:

uWSGI Go support (1.4 only)

goroutines

Argument: *setup goroutines*

A shortcut setting optimal options for goroutine-based apps, takes the number of goroutines to spawn as argument.

3.13.41 HTTP

See also:

Native HTTP support

http

Argument: *address*

Enable the embedded HTTP router/server/gateway/loadbalancer/proxy on the specified address.

http-processes, http-workers

Argument: number

Set the number of http processes to spawn.

http-var

Argument: string

Add a key=value item to the generated uwsgi packet.

This option may be declared multiple times.

http-to

Argument: string

Forward requests to the specified node (you can specify it multiple time for lb).

This option may be declared multiple times.

http-zerg

Argument: *corerouter zerg*

Attach the http router to a zerg server.

http-fallback

Argument: string

Fallback to the specified node in case of error.

This option may be declared multiple times.

http-modifier1

Argument: number

Set uwsgi protocol modifier1.

http-use-cache

Argument: no argument

Use uWSGI cache as key->value virtualhost mapper.

http-use-pattern

Argument: *corerouter use pattern*

Use the specified pattern for mapping requests to unix sockets.

http-use-base

Argument: *corerouter use base*

Use the specified base for mapping requests to unix sockets.

http-use-cluster

Argument: no argument

Load balance to nodes subscribed to the cluster.

http-events

Argument: number

Set the number of concurrent http async events.

http-subscription-server

Argument: *corerouter ss*

Enable the SubscriptionServer for clustering and massive hosting/load-balancing.

http-timeout

Argument: number

Set internal http socket timeout.

http-manage-expect

Argument: no argument

Manage the Expect HTTP request header.

http-keepalive

Argument: no argument

Support HTTP keepalive (non-pipelined) requests (requires backend support).

http-raw-body

Argument: no argument

Blindly send HTTP body to backends (required for WebSockets and Icecast support).

http-use-code-string

Argument: *corerouter cs*

Use code string as hostname->server mapper for the http router.

http-use-socket

Argument: optional *corerouter use socket*

Forward request to the specified uwsgi socket.

http-gracetime

Argument: number

Retry connections to dead static nodes after the specified amount of seconds.

http-quiet

Argument: no argument

Do not report failed connections to instances.

http-cheap

Argument: no argument

Run the http router in cheap mode.

http-stats, http-stats-server, http-ss

Argument: string

Run the http router stats server.

http-harakiri

Argument: number

Enable http router harakiri.

http-modifier2

Argument: number

Set uwsgi protocol modifier2.

http-auto-chunked

Argument: no argument

Automatically transform output to chunked encoding during HTTP 1.1 keepalive (if needed).

http-auto-gzip

Argument: no argument

Automatically gzip content if uWSGI-Encoding header is set to gzip, but content size (Content-Length/Transfer-Encoding) and Content-Encoding are not specified.

http-websockets

Argument: no argument

Automatically detect websockets connections and put the session in raw mode.

http-stud-prefix

Argument: *add addr list*

Expect a stud prefix (1byte family + 4/16 bytes address) on connections from the specified address.

3.13.42 HTTPS

See also:

HTTPS support (from 1.3)

https

Argument: *https config*

Add an https router/server on the specified address with specified certificate and key.

https-export-cert

Argument: no argument

Export uwsgi variable HTTPS_CC containing the raw client certificate.

http-to-https

Argument: *address*

Add an HTTP router/server on the specified address and redirect all of the requests to HTTPS.

https2

Argument: *https2*

Add an https/spdy router/server using keyval options.

https-session-context

Argument: string

Set the session id context to the specified value.

3.13.43 JVM

See also:

JVM in the uWSGI server (updated to 1.9)

jvm-main-class

Argument: string

Load the specified class and call its main() function.

jvm-classpath

Argument: string

Add the specified directory to the classpath.

This option may be declared multiple times.

jvm-opt

Argument: string

Add the specified jvm option.

This option may be declared multiple times.

jvm-class

Argument: string

Load the specified class.

This option may be declared multiple times.

jwsgi

Argument: string

Load the specified JWSGI application (syntax class:method).

3.13.44 Lua

See also:

Using Lua/WSAPI with uWSGI

lua

Argument: string

Load lua wsapi app.

lua-load

Argument: string

Load a lua file.

This option may be declared multiple times.

lua-shell, luashell

Argument: no argument

Run the lua interactive shell (debug.debug()).

3.13.45 Mono

See also:

The Mono ASP.NET plugin

mono-app

Argument: string

Load a Mono asp.net app from the specified directory.

This option may be declared multiple times.

mono-gc-freq

Argument: number

Run the Mono GC every <n> requests (default: run after every request).

mono-key

Argument: string

Select the ApplicationHost based on the specified CGI var.

This option may be declared multiple times.

mono-version

Argument: string

Set the Mono jit version.

mono-config

Argument: string

Set the Mono config file.

mono-assembly

Argument: string

Load the specified main assembly (default: uwsgi.dll).

mono-exec

Argument: string

Exec the specified assembly just before app loading.

This option may be declared multiple times.

mono-index

Argument: string

Add an asp.net index file.

This option may be declared multiple times.

3.13.46 Nagios output

See also:

Monitoring uWSGI with Nagios

nagios

Argument: no argument

Output Nagios-friendly status check information.

3.13.47 PAM

See also:

PAM

pam

Argument: string

Set the pam service name to use.

pam-user

Argument: string

Set a fake user for pam.

3.13.48 Perl

See also:

uWSGI Perl support (PSGI)

psgi

Argument: string

Load a psgi app.

perl-no-die-catch

Argument: no argument

Do not catch $SIG{__DIE__}.

perl-local-lib

Argument: string

Set perl locallib path.

perl-version

Argument: no argument

Print perl version.

perl-args

Argument: string

Add items (space separated) to @ARGV.

perl-arg

Argument: string

Add an item to @ARGV.

This option may be declared multiple times.

perl-exec

Argument: string

Exec the specified perl file before fork().

This option may be declared multiple times.

perl-exec-post-fork

Argument: string

Exec the specified perl file after fork().

This option may be declared multiple times.

perl-auto-reload

Argument: number

Enable perl auto-reloader with the specified frequency.

perl-auto-reload-ignore

Argument: string

Ignore the specified files when auto-reload is enabled.

This option may be declared multiple times.

3.13.49 PHP

See also:

Running PHP scripts in uWSGI

php-ini, php-config

Argument: *php ini*

Use this PHP.ini.

php-ini-append, php-config-append

Argument: string

Append this (these) php.inis to the first one.

This option may be declared multiple times.

php-set

Argument: *key=value*

Set a php config directive.

This option may be declared multiple times.

php-index

Argument: string

Set the file to open (like index.php) when a directory is requested.

This option may be declared multiple times.

php-docroot

Argument: string

Force php DOCUMENT_ROOT.

php-allowed-docroot

Argument: string

Add an allowed document root. Only scripts under these directories will be executed.

This option may be declared multiple times.

php-allowed-ext

Argument: string

Add an allowed php file extension. Only scripts ending with these extensions will run.

This option may be declared multiple times.

php-server-software

Argument: string

Force the SERVER_SOFTWARE value reported to PHP.

php-app

Argument: string

Run _only_ this file whenever a request to the PHP plugin is made.

php-dump-config

Argument: no argument

Dump php config (even if modified via –php-set or append options).

php-allowed-script

Argument: string

List the allowed php scripts (require absolute path).

This option may be declared multiple times.

php-app-qs

Argument: string

When in app mode force QUERY_STRING to the specified value + REQUEST_URI.

php-fallback

Argument: string

Run the specified php script when the request one does not exist.

php-app-bypass

Argument: *add regexp list*

If the regexp matches the uri the –php-app is bypassed.

php-var

Argument: string

Add/overwrite a CGI variable at each request.

This option may be declared multiple times.

3.13.50 Ping

See also:

`Ping`

ping

Argument: string

Ping specified uwsgi host.

If the ping is successful the process exits with a 0 code, otherwise with a value > 0.

ping-timeout

Argument: number **Default:** 3

Set ping timeout.

The maximum number of seconds to wait before considering a uWSGI instance dead

3.13.51 PyPy

pypy-lib

Argument: string

Set the path/name of the pypy library.

pypy-setup

Argument: string

Set the path of the python setup script.

pypy-home

Argument: string

Set the home of pypy library.

pypy-wsgi

Argument: string

Load a WSGI module.

pypy-wsgi-file

Argument: string

Load a WSGI/mod_wsgi file.

pypy-eval

Argument: string

Evaluate pypy code before fork().

This option may be declared multiple times.

pypy-eval-post-fork

Argument: string

Evaluate pypy code soon after fork().

This option may be declared multiple times.

pypy-exec

Argument: string

Execute pypy code from file before fork().

This option may be declared multiple times.

pypy-exec-post-fork

Argument: string

Execute pypy code from file soon after fork().

This option may be declared multiple times.

pypy-pp, pypy-python-path, pypy-pythonpath

Argument: string

Add an item to the pythonpath.

This option may be declared multiple times.

3.13.52 Python

See also:

Python support

wsgi-file, file

Argument: string

Load .wsgi file as the Python application.

eval

Argument: string

Evaluate Python code as WSGI entry point.

module, wsgi

Argument: string

Load a WSGI module as the application. The module (sans `.py`) must be importable, ie. be in `PYTHONPATH`.

This option may be set with `-w` from the command line.

callable

Argument: string **Default:** application

Set default WSGI callable name.

test

Argument: string

Test a module import.

This option may be set with `-J` from the command line.

home, virtualenv, venv, pyhome

Argument: string

Set PYTHONHOME/virtualenv.

This option may be set with `-H` from the command line.

See also:

`Virtualenv`

py-programname, py-program-name

Argument: string

Set python program name.

pythonpath, python-path, pp

Argument: *directory/glob*

Add directory (or an .egg or a glob) to the Python search path. This can be specified up to 64 times.

This option may be declared multiple times.

pymodule-alias

Argument: string

Add a python alias module.

This option may be declared multiple times.

See also:

`PyModuleAlias`

post-pymodule-alias

Argument: string

Add a python module alias after uwsgi module initialization.

This option may be declared multiple times.

import, pyimport, py-import, python-import

Argument: string

Import a python module.

This option may be declared multiple times.

shared-import, shared-pyimport, shared-py-import, shared-python-import

Argument: string

Import a python module in all of the processes.

This option may be declared multiple times.

spooler-import, spooler-pyimport, spooler-py-import, spooler-python-import

Argument: string

Import a python module in the spooler.

This option may be declared multiple times.

pyargv

Argument: string

Manually set `sys.argv` for python apps.

`pyargv="one two three"` will set `sys.argv` to `('one', 'two', 'three')`.

optimize

Argument: number

Set python optimization level (this may be dangerous for some apps).

This option may be set with `-O` from the command line.

paste

Argument: string

Load a paste.deploy config file.

See also:

`PythonPaste`

paste-logger

Argument: no argument

Enable paste fileConfig logger.

web3

Argument: string

Load a web3 app.

pump

Argument: string

Load a pump app.

wsgi-lite

Argument: string

Load a wsgi-lite app.

pecan

Argument: string

Load a pecan config file.

See also:

`PythonPecan`

ini-paste

Argument: *paste .INI*

Load a paste.deploy config file containing uwsgi section.

ini-paste-logged

Argument: *paste .INI*

Load a paste.deploy config file containing uwsgi section (load loggers too).

reload-os-env

Argument: no argument

Force reloading `os.environ` at each request.

no-site

Argument: no argument

Do not import the `site` module while initializing Python. This is usually only required for dynamic virtualenvs. If in doubt, do not enable.

pyshell

Argument: no argument

Run an interactive Python shell in the uWSGI environment.

pyshell-oneshot

Argument: no argument

Run an interactive Python shell in the uWSGI environment (one-shot variant).

python, py, pyrun

Argument: *.py file*

Run a Python script in the uWSGI environment.

py-tracebacker

Argument: string

Enable the uWSGI Python tracebacker.

py-auto-reload, py-autoreload, python-auto-reload, python-autoreload

Argument: number

Monitor Python modules' modification times to trigger reload (use only in development).

py-auto-reload-ignore

Argument: string

Ignore the specified module during auto-reload scan.

This option may be declared multiple times.

wsgi-env-behaviour, wsgi-env-behavior

Argument: string

Set the strategy for allocating/deallocating the WSGI env.

start_response-nodelay

Argument: no argument

Send WSGI http headers as soon as possible (PEP violation).

python-version

Argument: no argument

Report python version.

3.13.53 Rawrouter

See also:

Rawrouter

rawrouter

Argument: *corerouter*

Run the rawrouter on the specified port.

rawrouter-processes, rawrouter-workers

Argument: number

Prefork the specified number of rawrouter processes.

rawrouter-zerg

Argument: *corerouter zerg*

Attach the rawrouter to a zerg server.

rawrouter-use-cache

Argument: no argument

Use uWSGI cache as address->server mapper for the rawrouter.

rawrouter-use-pattern

Argument: *corerouter use pattern*

Use a pattern for rawrouter address->server mapping.

rawrouter-use-base

Argument: *corerouter use base*

Use a base dir for rawrouter address->server mapping.

rawrouter-fallback

Argument: string

Fallback to the specified node in case of error.

This option may be declared multiple times.

rawrouter-use-cluster

Argument: no argument

Load balance to nodes subscribed to the cluster.

rawrouter-use-code-string

Argument: *corerouter cs*

Use code string as address->server mapper for the rawrouter.

rawrouter-use-socket

Argument: optional *corerouter use socket*

Forward request to the specified uwsgi socket.

rawrouter-to

Argument: string

Forward requests to the specified uwsgi server (you can specify it multiple times for load balancing).

This option may be declared multiple times.

rawrouter-gracetime

Argument: number

Retry connections to dead static nodes after the specified amount of seconds.

rawrouter-events

Argument: number

Set the maximum number of concurrent events.

rawrouter-quiet

Argument: no argument

Do not report failed connections to instances.

rawrouter-cheap

Argument: no argument

Run the rawrouter in cheap mode.

rawrouter-subscription-server

Argument: *corerouter ss*

Run the rawrouter subscription server on the spcified address.

rawrouter-subscription-slot

Argument: *deprecated*

*** deprecated ***.

rawrouter-timeout

Argument: number

Set rawrouter timeout.

rawrouter-stats, rawrouter-stats-server, rawrouter-ss

Argument: string

Run the rawrouter stats server.

rawrouter-harakiri

Argument: number

Enable rawrouter harakiri.

rawrouter-max-retries

Argument: number

Set the maximum number of retries/fallbacks to other nodes.

3.13.54 Ring

ring-load, clojure-load

Argument: string

Load the specified clojure script.

This option may be declared multiple times.

ring-app

Argument: string

Map the specified ring application (syntax namespace:function).

3.13.55 RRDtool

See also:

RRDtool

rrdtool

Argument: string

Collect request data in the specified rrd file.

This option may be declared multiple times.

rrdtool-freq

Argument: number

Set collect frequency.

rrdtool-max-ds

Argument: number

Set maximum number of data sources.

rrdtool-lib

Argument: string

Set the name of rrd library (default: librrd.so).

3.13.56 Rsyslog

rsyslog-packet-size

Argument: number

Set maximum packet size for syslog messages (default 1024) WARNING! using packets > 1024 breaks RFC 3164 (#4.1).

rsyslog-split-messages

Argument: no argument

Split big messages into multiple chunks if they are bigger than allowed packet size (default is false).

3.13.57 Ruby

See also:

Ruby support

rails

Argument: string

Load a Ruby on Rails <= 2.x app.

rack

Argument: string

Load a Rack app.

ruby-gc-freq, rb-gc-freq

Argument: number

Set Ruby GC frequency.

rb-require, ruby-require, rbrequire, rubyrequire, require

Argument: string

Import/require a Ruby module/script.

This option may be declared multiple times.

shared-rb-require, shared-ruby-require, shared-rbrequire, shared-rubyrequire, shared-require

Argument: string

Import/require a Ruby module/script (shared).

This option may be declared multiple times.

gemset, rvm

Argument: string

Load the specified gemset (rvm).

rvm-path

Argument: string

Search for rvm in the specified directory.

This option may be declared multiple times.

rbshell

Argument: optional no argument

Run a Ruby/irb shell.

rb-threads, rbthreads, ruby-threads

Argument: number

Set the number of Ruby threads to run (Ruby 1.9+).

rb-lib, ruby-lib

Argument: string

Add a directory to the ruby libdir search path.

This option may be declared multiple times.

rbshell-oneshot

Argument: no argument

Set ruby/irb shell (one shot).

3.13.58 SSL Router

sslrouter

Argument: *sslrouter*

Run the sslrouter on the specified port.

sslrouter2

Argument: *sslrouter2*

Run the sslrouter on the specified port (key-value based).

sslrouter-session-context

Argument: string

Set the session id context to the specified value.

sslrouter-processes, sslrouter-workers

Argument: number

Prefork the specified number of sslrouter processes.

sslrouter-zerg

Argument: *corerouter zerg*

Attach the sslrouter to a zerg server.

sslrouter-use-cache

Argument: optional string

Use uWSGI cache as hostname->server mapper for the sslrouter.

sslrouter-use-pattern

Argument: *corerouter use pattern*

Use a pattern for sslrouter hostname->server mapping.

sslrouter-use-base

Argument: *corerouter use base*

Use a base dir for sslrouter hostname->server mapping.

sslrouter-fallback

Argument: string

Fallback to the specified node in case of error.

This option may be declared multiple times.

sslrouter-use-code-string

Argument: *corerouter cs*

Use code string as hostname->server mapper for the sslrouter.

sslrouter-use-socket

Argument: optional *corerouter use socket*

Forward request to the specified uwsgi socket.

sslrouter-to

Argument: string

Forward requests to the specified uwsgi server (you can specify it multiple times for load balancing).

This option may be declared multiple times.

sslrouter-gracetime

Argument: number

Retry connections to dead static nodes after the specified amount of seconds.

sslrouter-events

Argument: number

Set the maximum number of concusrent events.

sslrouter-max-retries

Argument: number

Set the maximum number of retries/fallbacks to other nodes.

sslrouter-quiet

Argument: no argument

Do not report failed connections to instances.

sslrouter-cheap

Argument: no argument

Run the sslrouter in cheap mode.

sslrouter-subscription-server

Argument: *corerouter ss*

Run the sslrouter subscription server on the spcified address.

sslrouter-timeout

Argument: number

Set sslrouter timeout.

sslrouter-stats, sslrouter-stats-server, sslrouter-ss

Argument: string

Run the sslrouter stats server.

sslrouter-harakiri

Argument: number

Enable sslrouter harakiri.

sslrouter-sni

Argument: no argument

Use SNI to route requests.

3.13.59 Symcall

symcall

Argument: string

Load the specified C symbol as the symcall request handler.

symcall-register-rpc

Argument: string

Load the specified C symbol as an RPC function (syntax: name function).

This option may be declared multiple times.

symcall-post-fork

Argument: string

Call the specified C symbol after each fork().

This option may be declared multiple times.

3.13.60 WebDAV

webdav-mount

Argument: string

Map a filesystem directory as a webdav store.

This option may be declared multiple times.

webdav-css

Argument: string

Add a css url for automatic webdav directory listing.

This option may be declared multiple times.

webdav-javascript, **webdav-js**

Argument: string

Add a javascript url for automatic webdav directory listing.

This option may be declared multiple times.

webdav-class-directory

Argument: string

Set the css directory class for automatic webdav directory listing.

webdav-div

Argument: string

Set the div id for automatic webdav directory listing.

webdav-lock-cache

Argument: string

Set the cache to use for webdav locking.

webdav-principal-base

Argument: string

Enable WebDAV Current Principal Extension using the specified base.

webdav-add-option

Argument: string

Add a WebDAV standard to the OPTIONS response.

This option may be declared multiple times.

webdav-add-prop

Argument: string

Add a WebDAV property to all resources.

This option may be declared multiple times.

webdav-add-collection-prop

Argument: string

Add a WebDAV property to all collections.

This option may be declared multiple times.

webdav-add-object-prop

Argument: string

Add a WebDAV property to all objects.

This option may be declared multiple times.

webdav-add-prop-href

Argument: string

Add a WebDAV property to all resources (href value).

This option may be declared multiple times.

webdav-add-collection-prop-href

Argument: string

Add a WebDAV property to all collections (href value).

This option may be declared multiple times.

webdav-add-object-prop-href

Argument: string

Add a WebDAV property to all objects (href value).

This option may be declared multiple times.

webdav-add-prop-comp

Argument: string

Add a WebDAV property to all resources (xml value).

This option may be declared multiple times.

webdav-add-collection-prop-comp

Argument: string

Add a WebDAV property to all collections (xml value).

This option may be declared multiple times.

webdav-add-object-prop-comp

Argument: string

Add a WebDAV property to all objects (xml value).

This option may be declared multiple times.

webdav-add-rtype-prop

Argument: string

Add a WebDAV resourcetype property to all resources.

This option may be declared multiple times.

webdav-add-rtype-collection-prop

Argument: string

Add a WebDAV resourcetype property to all collections.

This option may be declared multiple times.

webdav-add-rtype-object-prop

Argument: string

Add a WebDAV resourcetype property to all objects.

This option may be declared multiple times.

webdav-skip-prop

Argument: string

Do not add the specified prop if available in resource xattr.

This option may be declared multiple times.

3.13.61 XSLT

xslt-docroot

Argument: string

Add a document_root for xslt processing.

This option may be declared multiple times.

xslt-ext

Argument: string

Search for xslt stylesheets with the specified extension.

This option may be declared multiple times.

xslt-var

Argument: string

Get the xslt stylesheet path from the specified request var.

This option may be declared multiple times.

xslt-stylesheet

Argument: string

If no xslt stylesheet file can be found, use the specified one.

This option may be declared multiple times.

xslt-content-type

Argument: string

Set the content-type for the xslt rsult (default: text/html).

3.14 Defining new options for your instances

Sometimes the built-in options are not enough. For example, you may need to give your customers custom options for configuring their apps on your platform. Or you need to configure so many instances you want to simplify things such as per-datacenter or per-server-type options. Declaring new options for your config files/command-line is a good way of achieving these goals.

To define new options use `--declare-option`:

```
--declare-option <option_name>=<option1=value1>[;<option2=value2>;<option3=value3>...]
```

An useful example could be defining a "redirect" option, using the redirect plugin of the InternalRouting subsystem:

```
--declare-option "redirect=route=\$1 redirect:\$2"
```

This will declare a new option called `redirect` that takes 2 arguments. Those arguments will be expanded using the $-prefixed variables. Like shell scripts, *the backslash is required to make your shell not expand these values*.

Now you will be able to define a redirect in your config files:

```
uwsgi --declare-option "redirect=route=\$1 redirect:\$2" --ini config.ini
```

Config.ini:

```
[uwsgi]
socket = :3031
; define my redirects
redirect = ^/foo http://unbit.it
redirect = \.jpg$ http://uwsgi.it/test
redirect = ^/foo/bar/ /test
```

or directly on the command line:

```
uwsgi --declare-option "redirect=route=\$1 redirect:\$2" --socket :3031 --redirect "^/foo http://unbi
```

3.14.1 More fun: a bunch of shortcuts

Now we will define new options for frequently-used apps.

Shortcuts.ini:

```
[uwsgi]
; let's define a shortcut for trac (new syntax: trac=<path_to_trac_instance>)
declare-option = trac=plugin=python;env=TRAC_ENV=$1;module=trac.web.main:dispach_request
; one for web2py (new syntax: web2py=<path_to_web2_py_dir>)
declare-option = web2py=plugin=python;chdir=$1;module=wsgihandler
; another for flask (new syntax: flask=<path_to_your_app_entry_point>)
declare-option = flask=plugin=python;wsgi-file=$1;callable=app
```

To hook up a Trac instance on /var/www/trac/fooenv:

```
[uwsgi]
; include new shortcuts
ini = shortcuts.ini

; classic options
http = :8080
master = true
threads = 4

; our new option
trac = /var/www/trac/fooenv
```

A config for Web2py, in XML:

```
<uwsgi>
  <!-- import shortcuts -->
  <ini>shortcuts.ini</ini>
  <!-- run the https router with HIGH ciphers -->
  <https>:443,test.crt,test.key,HIGH</https>

  <master/>
  <processes>4</processes>

  <!-- load web2py from /var/www/we2py -->
  <web2py>/var/www/we2py</web2py>
</uwsgi>
```

3.14.2 A trick for the Emperor: automatically import shortcuts for your vassals

If you manage your customers/users with the *Emperor*, you can configure it to automatically import your shortcuts in each vassal.

```
uwsgi --emperor /etc/uwsgi/vassals --vassals-include /etc/uwsgi/shortcuts.ini
```

For multiple shortcuts use:

```
uwsgi --emperor /etc/uwsgi/vassals --vassals-include /etc/uwsgi/shortcuts.ini --vassals-include /etc/
```

Or (with a bit of *configuration logic magic*):

```
[uwsgi]
emperor = /etc/uwsgi/vassals

for = shortcuts shortcuts2 shortcuts3
  vassals-include = /etc/uwsgi/%(_).ini
endfor =
```

3.14.3 An advanced trick: embedding shortcuts in your uWSGI binary

uWSGI's build system allows you to embed files, be they generic files or configuration, in the server binary. Abusing this feature will enable you to embed your new option shortcuts to the server binary, automagically allowing users to use them. To embed your shortcuts file, edit your build profile (like `buildconf/base.ini`) and set `embed_config` to the path of the shortcuts file. Rebuild your server and your new options will be available.

See also:

`BuildConf`

3.15 How uWSGI parses config files

Until uWSGI 1.1 the parsing order has not been 'stable' or 'reliable'.

Starting from uWSGI 1.1 (thanks to its new options subsystem) we have a general rule: top-bottom and expand asap.

Top-bottom means options are internally ordered as they are parsed, while "expand asap" means to inject the options of a requested config file, interrupting the currently parsed one:

Note that the `inherit` option behaves differently from the other inlcude options: It is expanded *after* variable expansion, so any environment variables, external files and placeholders are *not* expanded. Magic variables (*e.g.* `%n`) are expanded normally.

file1.ini (the one requested from the command line)

```
[uwsgi]
socket = :3031
ini = file2.ini
socket = :3032
chdir = /var/www
```

file2.ini

```
[uwsgi]
master = true
memory-report = true
processes = 4
```

internally will be assembled in:

```
[uwsgi]
socket = :3031
ini = file2.ini
master = true
memory-report = true
processes = 4
socket = :3032
chdir = /var/www
```

A more complex example:

file1.ini (the one requested from the command line)

```
[uwsgi]
socket = :3031
ini = file2.ini
socket = :3032
chdir = /var/www
```

file2.ini

```
[uwsgi]
master = true
xml = file3.xml
memory-report = true
processes = 4
```

file3.xml

```
<uwsgi>
  <plugins>router_uwsgi</plugins>
```

```
  <route>^/foo uwsgi:127.0.0.1:4040,0,0</route>
</uwsgi>
```

will result in:

```
[uwsgi]
socket = :3031
ini = file2.ini
master = true
xml = file3.xml
plugins = router_uwsgi
route = ^/foo uwsgi:127.0.0.1:4040,0,0
memory-report = true
processes = 4
socket = :3032
chdir = /var/www
```

3.15.1 Expanding variables/placeholders

After the internal config tree is assembled, variables and placeholder substitution will be applied.

The first step is substituting all of the $(VALUE) occurence with the value of the environment variable VALUE

```
[uwsgi]
foobar = $(PATH)
```

foobar value will be the content of shell's PATH variable

The second step will expand text files embraced in @(FILENAME)

```
[uwsgi]
nodename = @(/etc/hostname)
```

nodename value will be the content of /etc/hostname

The last step is placeholder substitution. A placeholder is a reference to another option:

```
[uwsgi]
socket = :3031
foobar = %(socket)
```

the content of foobar will be mapped to the content of socket

3.15.2 A note on magic variables

Config files, support another form of variables, called 'magic'. As they refer to the config file itself, they will be parsed asap:

```
[uwsgi]
my_config_file = %p
```

the content of my_config_file will be set to %p value (the current file absolute path) as soon as it is parsed. That means %p (or whatever magic vars you need) will be always consistent in the currently parsing config file.

3.16 uwsgi protocol magic variables

You can dynamically tune or configure various aspects of the uWSGI server using special variables passed by the web server (or in general by a uwsgi compliant client).

- For Nginx, the `uwsgi_param <name> <value>;` directive is used.
- For Apache, the `SetEnv <name> <value>` directive is used.

3.16.1 UWSGI_SCHEME

Set the URL scheme when it cannot be reliably determined. This may be used to force HTTPS (with the value `https`), for instance.

3.16.2 UWSGI_SCRIPT

Load the specified script as a new application mapped to `SCRIPT_NAME`. The app will be obviously only be loaded once, not on each request.

```
uwsgi_param UWSGI_SCRIPT werkzeug.testapp:test_app;
uwsgi_param SCRIPT_NAME  /testapp;
```

3.16.3 UWSGI_MODULE and UWSGI_CALLABLE

Load a new app (defined as `module:callable`) mapped into `SCRIPT_NAME`.

```
uwsgi_param UWSGI_MODULE werkzeug.testapp;
uwsgi_param UWSGI_CALLABLE test_app;
uwsgi_param SCRIPT_NAME  /testapp;
```

3.16.4 UWSGI_PYHOME

Dynamically set the Python *Virtualenv support* for a *dynamic application*.

See also:

DynamicVirtualenv

3.16.5 UWSGI_CHDIR

`chdir()` to the specified directory before managing the request.

3.16.6 UWSGI_FILE

Load the specified file as a new dynamic app.

3.16.7 UWSGI_TOUCH_RELOAD

Reload the uWSGI stack when the specified file's modification time has changed since the last request.

```
location / {
  include uwsgi_params;
  uwsgi_param UWSGI_TOUCH_RELOAD /tmp/touchme.foo;
  uwsgi_pass /tmp/uwsgi.sock;
}
```

3.16.8 UWSGI_CACHE_GET

See also:

The uWSGI caching framework

Check the uWSGI cache for a specified key. If the value is found, it will be returned as raw HTTP output instead of the usual processing of the request.

```
location / {
  include uwsgi_params;
  uwsgi_param UWSGI_CACHE_GET $request_uri;
  uwsgi_pass 127.0.0.1:3031;
}
```

3.16.9 UWSGI_SETENV

Set the specified environment variable for a new dynamic app.

Note: To allow this in Python applications you need to enable the `reload-os-env` uWSGI option.

Dynamically load a Django app without using a WSGI file/module:

```
location / {
  include uwsgi_params;
  uwsgi_param UWSGI_SCRIPT django.core.handlers.wsgi:WSGIHandler();
  uwsgi_param UWSGI_CHDIR /mydjangoapp_path;
  uwsgi_param UWSGI_SETENV DJANGO_SETTINGS_MODULE=myapp.settings;
}
```

3.16.10 UWSGI_APPID

Note: Available since 0.9.9.

Bypass `SCRIPT_NAME` and `VirtualHosting` to let the user choose the mountpoint without limitations (or headaches).

The concept is very generic: `UWSGI_APPID` is the identifier of an application. If it is not found in the internal list of apps, it will be loaded.

```
server {
    server_name server001;
    location / {
        include uwsgi_params;
```

```
        uwsgi_param UWSGI_APPID myfunnyapp;
        uwsgi_param UWSGI_FILE /var/www/app1.py
    }
}

server {
    server_name server002;
    location / {
        include uwsgi_params;
        uwsgi_param UWSGI_APPID myamazingapp;
        uwsgi_param UWSGI_FILE /var/www/app2.py
    }
}
```

3.17 The uwsgi Protocol

The uwsgi (lowercase!) protocol is the native protocol used by the uWSGI server.

It is a binary protocol that can carry any type of data. The first 4 bytes of a uwsgi packet describe the type of the data contained by the packet.

Every uwsgi request generates an response in the uwsgi format.

Even the web server handlers obey this rule, as an HTTP response is a valid uwsgi packet (look at the `modifier1` = 72).

The protocol works mainly via TCP but the master process can bind to a UDP Unicast/Multicast for *The embedded SNMP server* or cluster management/messaging requests.

SCTP support is being worked on.

3.17.1 uwsgi packet header

```
struct uwsgi_packet_header {
    uint8_t modifier1;
    uint16_t datasize;
    uint8_t modifier2;
};
```

Unless otherwise specified the `datasize` value contains the size (16-bit little endian) of the packet body.

3.17.2 Packet descriptions

modifier1	datasize	modifier2
0	size of WSGI block vars (HTTP request body excluded)	0
1	reserved for UNBIT	
2	reserved for UNBIT	
3	reserved for UNBIT	
5	size of *PSGI* block vars (HTTP request body excluded)	0
6	size of LUA WSAPI block vars (HTTP request body excluded)	0
7	size of RACK block vars (HTTP request body excluded)	0
8	size of JWSGI/Ring block vars (HTTP request body excluded)	0

modifier1	datasize	modifier2
9	size of CGI block vars (HTTP request body excluded)	0
10	size of block vars	0- 255
14	size of CGI block vars (HTTP request body excluded)	0
15	size of Mono ASP.NET block vars (HTTP request body excluded)	0
17	size of Spooler block vars	0- 255
18	size of CGI block vars	0-255
22	size of code string	0- 255
23	size of CGI vars	0- 255
24	size of CGI vars	0- 255
25	size of CGI vars	0- 255
26	size of CGI vars	0- 255
27	0	0- 255
28	0	0- 255
30	size of WSGI block vars (HTTP request body excluded)	0 (if defined the size of the block vars is 24bit l
31	size of block vars	0- 255
32	size of char array	0- 255
33	size of marshal object	0- 255
48	snmp specific	snmp specific
72	chr(TT)	chr(P)
73	announce message size (for sanity check)	announce type (0 = hostname)
74	multicast message size (for sanity check)	0
95	cluster membership dict size	action
96	log message size	0
97	0	0, 1
98	0	0, 1
99	size of options dictionary (if response)	0, 1
100	0	0, 1
101	size of packet	0
109	size of clean payload	0 to 255
110	size of payload	0 to 255
111	size of packet	0, 1, 2, 3
173	size of packet	0, 1
200	0	0
224	size of packet	0
255	0	0- 255

3.17.3 The uwsgi vars

The uwsgi block vars represent a dictionary/hash. Every key-value is encoded in this way:

```
struct uwsgi_var {
    uint16_t key_size;
    uint8_t key[key_size];
    uint16_t val_size;
    uint8_t val[val_size];
}
```

3.18 Managing external daemons/services

uWSGI can easily monitor external processes, allowing you to increase reliability and usability of your multi-tier apps. For example you can manage services like Memcached, Redis, Celery, Ruby delayed_job or even dedicated postgresql instances.

3.18.1 Kinds of services

Currently uWSGI supports 3 categories of processes:

- `--attach-daemon` – directly attached non daemonized processes
- `--smart-attach-daemon` – pidfile governed (both foreground and daemonized)
- `--smart-attach-daemon2` – pidfile governed with daemonization management

The first category allows you to directly attach processes to the uWSGI master. When the master dies or is reloaded these processes are destroyed. This is the best choice for services that must be flushed whenever the app is restarted.

Pidfile governed processes can survive death or reload of the master so long as their pidfiles are available and the pid conteined therein matches a running pid. This is the best choice for processes requiring longer persistence, and for which a brutal kill could mean loss of data such as a database.

The last category is an superset of the second one. If your process does not support daemonization or writing to pidfile, you can let the master do the management. Very few daemons/applications require this feature, but it could be useful for tiny prototype applications or simply poorly designed ones.

Since uWSGI 2.0 a fourth option `--attach-daemon2` has been added for advanced configurations (see below)

3.18.2 Examples

Managing a **memcached** instance in 'dumb' mode. Whenever uWSGI is stopped or reloaded, memcached is destroyed.

```
[uwsgi]
master = true
socket = :3031
attach-daemon = memcached -p 11311 -u roberto
```

Managing a **memcached** instance in 'smart' mode. Memcached survives uWSGI stop and reload.

```
[uwsgi]
master = true
socket = :3031
smart-attach-daemon = /tmp/memcached.pid memcached -p 11311 -d -P /tmp/memcached.pid -u roberto
```

Managing 2 **mongodb** instances in smart mode:

```
[uwsgi]
master = true
socket = :3031
smart-attach-daemon = /tmp/mongo1.pid mongod --pidfilepath /tmp/mongo1.pid --dbpath foo1 --port 5000
smart-attach-daemon = /tmp/mongo2.pid mongod --pidfilepath /tmp/mongo2.pid --dbpath foo2 --port 5000
```

Managing **PostgreSQL** dedicated-instance (cluster in /db/foobar1):

```
[uwsgi]
master = true
socket = :3031
smart-attach-daemon = /db/foobar1/postmaster.pid /usr/lib/postgresql/9.1/bin/postgres -D /db/foobar1
```

Managing **celery**:

```
[uwsgi]
master = true
socket = :3031
smart-attach-daemon = /tmp/celery.pid celery -A tasks worker --pidfile=/tmp/celery.pid
```

Managing **delayed_job**:

```
[uwsgi]
master = true
socket = :3031
env = RAILS_ENV=production
rbrequire = bundler/setup
rack = config.ru
chdir = /var/apps/foobar
smart-attach-daemon = %(chdir)/tmp/pids/delayed_job.pid %(chdir)/script/delayed_job start
```

Managing **dropbear**:

```
[uwsgi]
namespace = /ns/001/:testns
namespace-keep-mount = /dev/pts
socket = :3031
exec-as-root = chown -R www-data /etc/dropbear
attach-daemon = /usr/sbin/dropbear -j -k -p 1022 -E -F -I 300
```

When using the namespace option you can attach dropbear daemon to allow direct access to the system inside the specified namespace. This requires the */dev/pts* filesystem to be mounted inside the namespace, and the user your workers will be running as have access to the */etc/dropbear* directory inside the namespace.

3.18.3 Legion support

Starting with uWSGI 1.9.9 it's possible to use *The uWSGI Legion subsystem* subsystem for daemon management. Legion daemons will will be executed only on the legion lord node, so there will always be a single daemon instance running in each legion. Once the lord dies a daemon will be spawned on another node. To add a legion daemon use –legion-attach-daemon, –legion-smart-attach-daemon and –legion-smart-attach-daemon2 options, they have the same syntax as normal daemon options. The difference is the need to add legion name as first argument.

Example:

Managing **celery beat**:

```
[uwsgi]
master = true
socket = :3031
legion-mcast = mylegion 225.1.1.1:9191 90 bf-cbc:mysecret
legion-smart-attach-daemon = mylegion /tmp/celery-beat.pid celery beat --pidfile=/tmp/celery-beat.pi
```

3.18.4 –attach-daemon2

This option has been added in uWSGI 2.0 and allows advanced configurations. It is a keyval option, and it accepts the following keys:

`command` the command line to execute

`cmd` alias for command

`exec` alias for command

`freq` maximum attempts before considering a daemon "broken"

`pidfile` the pidfile to check (enable smart mode)

`control` if set, the daemon became a 'control' one: if it dies the whole uWSGI instance dies

`daemonize` daemonize the process (enable smart2 mode)

`daemon` alias for daemonize

`touch` semicolon separated list of files to check: whenever they are 'touched', the daemon is restarted

`stopsignal` the signal number to send to the daemon when uWSGI is stopped

`stop_signal` alias for stopsignal

`reloadsignal` the signal to send to the daemon when uWSGI is reloaded

`reload_signal` alias for reloadsignal

`stdin` if set the file descriptor zero is not remapped to /dev/null

`uid` drop privileges to the specified uid (requires master running as root)

`gid` drop privileges to the specified gid (requires master running as root)

`ns_pid` spawn the process in a new pid namespace (requires master running as root, Linux only)

Example:

```
[uwsgi]
attach-daemon2 = cmd=my_daemon.sh,pidfile=/tmp/my.pid,uid=33,gid=33,stopsignal=3
```

3.19 The Master FIFO

Available from uWSGI 1.9.17

You can tell the master to create a UNIX named pipe (fifo) you can use to issue commands to the master.

Generally you use UNIX signals to manage the master, but we are run out of signals numbers and (more important) not needing to mess with pids simplify the implementation of external management scripts.

To create the fifo just add `--master-fifo <filename>` then start issuing commands to it:

```
echo r > /tmp/yourfifo
```

you can send multiple commands in one shot:

```
echo +++s > /tmp/yourfifo
```

will add 3 workers and will print stats

3.19.1 Available commands

- '0' to '9' set the fifo slot (see below)
- '-' decrease the number of workers when in cheaper mode (add `--cheaper-algo manual` for full control)
- '+' increase the number of workers when in cheaper mode (add `--cheaper-algo manual` for full control)
- 'c' trigger chain reload
- 'C' set cheap mode
- 'E' trigger an Emperor rescan
- 'f' re-fork the master (dangerous, but very powerful)
- 'l' reopen log file (need –log-master and –logto/–logto2)
- 'L' trigger log rotation (need –log-master and –logto/–logto2)
- 'p' pause/resume the instance
- 'P' update pidfiles (can be useful after master re-fork)
- 'q' gracefully shutdown the instance
- 'Q' brutally shutdown the instance
- 'r' send graceful reload
- 'R' send brutal reload
- 's' print stats in the logs
- 'S' block/unblock subscriptions
- 'w' gracefully reload workers
- 'W' brutally reload workers

3.19.2 FIFO slots

uWSGI supports up to 10 different fifo files. By default the first specified is bound (mapped as '0').

During the whole instance lifetime you can change from one fifo file to another simply sending the number of the fifo slot to use:

```
[uwsgi]
master-fifo = /tmp/fifo0
master-fifo = /tmp/fifo1
master-fifo = /var/run/foofifo
processes = 2
...
```

By default /tmp/fifo0 will be allocated, but after sending:

```
echo 1 > /tmp/fifo0
```

the /tmp/fifo1 file will be bound

This is very useful to map fifo files to specific instance when you abuse the 'fork the master' command (the 'f' one):

```
echo 1fp > /tmp/fifo0
```

after sending this command, a new uwsgi instance (inheriting all of the bound sockets) will be spawned, the old one will be put in "paused" mode (p command).

As we have sent the '1' command before 'f' and 'p' the old instance will now accepts commands on /tmp/fifo1 (the slot 1) will the new one will use the default one (the '0')

There are lot of tricks you can accomplish, and lot of way to abuse the forking of the master. Just take in account that corner-case problems can raise all over the place, expecially if you use the most complex features of uWSGI.

3.19.3 Notes

The FIFO is created in non-blocking modes and recreated by the master every time a client disconnects.

You can override (or add) commands using the global array uwsgi_fifo_table via plugins or c hooks

Only the uid running the master has write access to the fifo

3.20 Socket activation with inetd/xinetd

Inetd and Xinetd are two daemons used to start network processes on demand. You can use this in uWSGI too.

3.20.1 Inetd

```
127.0.0.1:3031 stream tcp wait root /usr/bin/uwsgi uwsgi -M -p 4 --wsgi-file /root/uwsgi/welcome.py -
```

With this config you will run uWSGI on port 3031 as soon as the first connection is made. Note: the first argument (the one soon after /usr/bin/uwsgi) is mapped to `argv[0]`. Do not forget this – always set it to `uwsgi` if you want to be sure.

3.20.2 Xinetd

```
service uwsgi
    {
            disable         = no
            id              = uwsgi-000
            type            = UNLISTED
            socket_type     = stream
            server          = /root/uwsgi/uwsgi
            server_args     = --chdir /root/uwsgi/ --module welcome --logto /tmp/uwsgi.log
            port            = 3031
            bind            = 127.0.0.1
            user            = root
            wait            = yes
    }
```

Again, you do not need to specify the socket in uWSGI, as it will be passed to the server by xinetd.

3.21 Running uWSGI via Upstart

Upstart is the init system of Ubuntu-like distributions.

It is based on declarative configuration files – not shell scripts of yore – that are put in the `/etc/init` directory.

3.21.1 A simple script (/etc/init/uwsgi.conf)

```
# simple uWSGI script

description "uwsgi tiny instance"
start on runlevel [2345]
stop on runlevel [06]

exec uwsgi --master --processes 4 --die-on-term --socket :3031 --wsgi-file /var/www/myapp.wsgi
```

3.21.2 Using the Emperor

See also:

The uWSGI Emperor – multi-app deployment

A better approach than init files for each app would be to only start an Emperor via Upstart and let it deal with the rest.

```
# Emperor uWSGI script

description "uWSGI Emperor"
start on runlevel [2345]
stop on runlevel [06]

exec uwsgi --emperor /etc/uwsgi
```

If you want to run the Emperor under the master process (for accessing advanced features) remember to add –die-on-term

```
# Emperor uWSGI script

description "uWSGI Emperor"
start on runlevel [2345]
stop on runlevel [06]

exec uwsgi --master --die-on-term --emperor /etc/uwsgi
```

3.21.3 What is –die-on-term?

By default uWSGI maps the SIGTERM signal to “a brutal reload procedure”.

However, Upstart uses SIGTERM to completely shutdown processes. `die-on-term` inverts the meanings of SIGTERM and SIGQUIT to uWSGI.

The first will shutdown the whole stack, the second one will brutally reload it.

3.21.4 Socket activation (from Ubuntu 12.04)

Newer Upstart releases have an Inetd-like feature that lets processes start when connections are made to specific sockets.

You can use this feature to start uWSGI only when a client (or the webserver) first connects to it.

The ‘start on socket’ directive will trigger the behaviour.

You do not need to specify the socket in uWSGI as it will be passed to it by Upstart itself.

```
# simple uWSGI script

description "uwsgi tiny instance"
start on socket PROTO=inet PORT=3031
stop on runlevel [06]

exec uwsgi --master --processes 4 --die-on-term --wsgi-file /var/www/myapp.wsgi
```

3.22 SystemD

uWSGI is a new-style daemon for [http://www.freedesktop.org/wiki/Software/systemd systemd].

It can notify status change and readyness.

When uWSGI detects it is running under systemd, the notification system is enabled.

3.22.1 Adding the Emperor to systemd

The best approach to integrate uWSGI apps with your init system is using the *Emperor*-

Your init system will talk only with the Emperor that will rule all of the apps itself.

Create a systemd service file (you can save it as /etc/systemd/system/emperor.uwsgi.service)

```
[Unit]
Description=uWSGI Emperor
After=syslog.target

[Service]
ExecStart=/root/uwsgi/uwsgi --ini /etc/uwsgi/emperor.ini
Restart=always
KillSignal=SIGQUIT
Type=notify
StandardError=syslog
NotifyAccess=main

[Install]
WantedBy=multi-user.target
```

Then run it

```
systemctl start emperor.uwsgi.service
```

And check its status.

```
systemctl status emperor.uwsgi.service
```

You will see the Emperor reporting the number of governed vassals to Systemd (and to you).

```
emperor.uwsgi.service - uWSGI Emperor
 Loaded: loaded (/etc/systemd/system/emperor.uwsgi.service)
       Active: active (running) since Tue, 17 May 2011 08:51:31 +0200; 5s ago
Main PID: 30567 (uwsgi)
       Status: "The Emperor is governing 1 vassals"
       CGroup: name=systemd:/system/emperor.uwsgi.service
                30567 /root/uwsgi/uwsgi --ini /etc/uwsgi/emperor.ini
```

```
30568 /root/uwsgi/uwsgi --ini werkzeug.ini
30569 /root/uwsgi/uwsgi --ini werkzeug.ini
```

You can stop the Emperor (and all the apps it governs) with

```
systemctl stop emperor.uwsgi.service
```

A simple `emperor.ini` could look like this (www-data is just an anonymous user)

```
[uwsgi]
emperor = /etc/uwsgi/vassals
uid = www-data
gid = www-data
```

If you want to allow each vassal to run under different privileges, remove the `uid` and `gid` options from the emperor configuration (and please read the Emperor docs!)

3.22.2 Logging

Using the previous service file all of the Emperor messages go to the syslog. You can avoid it by removing the `StandardError=syslog` directive.

If you do that, be sure to set a `--logto` option in your Emperor configuration, otherwise all of your logs will be lost!

3.22.3 Putting sockets in /run/

On a modern system, /run/ is mounted as a tmpfs and is the right place to put sockets and pidfiles into. You can have systemd create a uwsgi directory to put them into by creating a systemd-tmpfiles configuration file (you can save it as /etc/tmpfiles.d/emperor.uwsgi.conf):

```
d /run/uwsgi 0755 www-data www-data -
```

3.22.4 Socket activation

Starting from uWSGI 0.9.8.3 socket activation is available. You can setup systemd to spawn uWSGI instances only after the first socket connection.

Create the required emperor.uwsgi.socket (in `/etc/systemd/system/emperor.uwsgi.socket`). Note that the *.socket file name must match the *.service file name.

```
[Unit]
Description=Socket for uWSGI Emperor

[Socket]
# Change this to your uwsgi application port or unix socket location
ListenStream=/tmp/uwsgid.sock

[Install]
WantedBy=sockets.target
```

Then disable the service and enable the socket unit.

```
# systemctl disable emperor.uwsgi.service
# systemctl enable emperor.uwsgi.socket
```

3.23 Running uWSGI instances with Circus

Circus (http://circus.readthedocs.org/en/0.7/) is a process manager written in Python. It is very similar to projects like Supervisor, but with several additional features. Although most, if not all, of it's functionalities have a counterpart in uWSGI, Circus can be used as a library allowing you to build dynamic configurations (and extend uWSGI patterns). This aspect is very important and may be the real selling point of Circus.

3.23.1 Socket activation

Based on the venerable inetd pattern, Circus can bind to sockets and pass them to children.

Start with a simple Circus config (call it circus.ini):

```
[circus]
endpoint = tcp://127.0.0.1:5555
pubsub_endpoint = tcp://127.0.0.1:5556
stats_endpoint = tcp://127.0.0.1:5557

[watcher:dummy]
cmd = uwsgi --http-socket fd://$(circus.sockets.foo) --wsgi-file yourapp.wsgi
use_sockets = True
send_hup = True
stop_signal = QUIT

[socket:foo]
host = 0.0.0.0
port = 8888
```

run it with

```
circusd circus.ini
```

3.23.2 (Better) Socket activation

If you want to spawn instances on demand, you will likely want to shut them down when they are no longer used. To accomplish that use the –idle uWSGI option.

```
[circus]
check_delay = 5
endpoint = tcp://127.0.0.1:5555
pubsub_endpoint = tcp://127.0.0.1:5556
stats_endpoint = tcp://127.0.0.1:5557

[watcher:dummy]
cmd = uwsgi --master --idle 60 --http-socket fd://$(circus.sockets.foo) --wsgi-file yourapp.wsgi
use_sockets = True
warmup_delay = 0
send_hup = True
stop_signal = QUIT

[socket:foo]
host = 0.0.0.0
port = 8888
```

This time we have enabled the master process. It will manage the –idle option, shutting down the instance if it is inactive for more than 60 seconds

3.24 Embedding an application in uWSGI

Starting from uWSGI 0.9.8.2, you can embed files in the server binary. These can be any file type, including configuration files. You can embed directories too, so by hooking the Python module loader you can transparently import packages, too. In this example we'll be embedding a full Flask project.

3.24.1 Step 1: creating the build profile

We're assuming you have your uWSGI source at the ready.

In the `buildconf` directory, define your profile – let's call it flask.ini:

```
[uwsgi]
inherit = default
bin_name = myapp
embed_files = bootstrap.py,myapp.py
```

`myapp.py` is a simple flask app.

```
from flask import Flask
app = Flask(__name__)
app.debug = True

@app.route('/')
def index():
    return "Hello World"
```

`bootstrap.py` is included in the source distribution. It will extend the python import subsystem to use files embedded in uWSGI.

Now compile your app-inclusive server. Files will be embedded as symbols in the executable. Dots and dashes, etc. in filenames are thus transformed to underscores.

```
python uwsgiconfig.py --build flask
```

As `bin_name` was `myapp`, you can now run

```
./myapp --socket :3031 --import sym://bootstrap_py --module myapp:app
```

The `sym://` pseudoprotocol enables uWSGI to access the binary's embedded symbols and data, in this case importing bootstrap.py directly from the binary image.

3.24.2 Step 2: embedding the config file

We want our binary to automatically load our Flask app without having to pass a long command line.

Let's create the configuration – flaskconfig.ini:

```
[uwsgi]
socket = 127.0.0.1:3031
import = sym://bootstrap_py
module = myapp:app
```

And add it to the build profile as a config file.

```
[uwsgi]
inherit = default
bin_name = myapp
embed_files = bootstrap.py,myapp.py
embed_config = flaskconfig.ini
```

Then, after you rebuild the server

```
python uwsgiconfig.py --build flask
```

you can now simply launch

```
./myapp
# Remember that this new binary continues to be able to take parameters and config files:
./myapp --master --processes 4
```

3.24.3 Step 3: embedding flask itself

Now, we are ready to kick asses with uWSGI ninja awesomeness. We want a single binary embedding all of the Flask modules, including Werkzeug and Jinja2, Flask's dependencies. We need to have these packages' directories and then specify them in the build profile.

```
[uwsgi]
inherit = default
bin_name = myapp
embed_files = bootstrap.py,myapp.py,werkzeug=site-packages/werkzeug,jinja2=site-packages/jinja2,flask
embed_config = flaskconfig.ini
```

Note: This time we have used the form "name=directory" to force symbols to a specific names to avoid ending up with a clusterfuck like `site_packages_flask___init___py`.

Rebuild and re-run. We're adding –no-site when running to show you that the embedded modules are being loaded.

```
python uwsgiconfig.py --build flask
./myapp --no-site --master --processes 4
```

3.24.4 Step 4: adding templates

Still not satisfied? WELL YOU SHOULDN'T BE.

```
[uwsgi]
inherit = default
bin_name = myapp
embed_files = bootstrap.py,myapp.py,werkzeug=site-packages/werkzeug,jinja2=site-packages/jinja2,flask
embed_config = flaskconfig.ini
```

Templates will be added to the binary... but we'll need to instruct Flask on how to load templates from the binary image by creating a custom Jinja2 template loader.

```
from flask import Flask, render_template
from flask.templating import DispatchingJinjaLoader

class SymTemplateLoader(DispatchingJinjaLoader):

    def symbolize(self, name):
```

```
        return name.replace('.','_').replace('/', '_').replace('-','_')

    def get_source(self, environment, template):
        try:
            import uwsgi
            source = uwsgi.embedded_data("templates_%s" % self.symbolize(template))
            return source, None, lambda: True
        except:
            pass
        return super(SymTemplateLoader, self).get_source(environment, template)

app = Flask(__name__)
app.debug = True

app.jinja_env.loader = SymTemplateLoader(app)

@app.route('/')
def index():
    return render_template('hello.html')

@app.route('/foo')
def foo():
    return render_template('bar/foo.html')
```

POW! BIFF! NINJA AWESOMENESS.

3.25 Logging

See also:

Formatting uWSGI requests logs

3.25.1 Basic logging

The most basic form of logging in uWSGI is writing requests, errors, and informational messages to stdout/stderr. This happens in the default configuration. The most basic form of log redirection is the `--logto` / `--logto2` / `--daemonize` options which allow you to redirect logs to files.

Basic logging to files

To log to files instead of stdout/stderr, use `--logto`, or to simultaneously daemonize uWSGI, `--daemonize`.

```
./uwsgi -s :3031 -w simple_app --daemonize /tmp/mylog.log
./uwsgi -s :3031 -w simple_app --logto /tmp/mylog.log
# logto2 only opens the log file after privileges have been dropped to the specified uid/gid.
./uwsgi -s :3031 -w simple_app --uid 1001 --gid 1002 --logto2 /tmp/mylog.log
```

Basic logging (connected UDP mode)

With UDP logging you can centralize cluster logging or redirect the persistence of logs to another machinei to offload disk I/O. UDP logging works in both daemonized and interactive modes. UDP logging operaties in connected-socket mode, so the UDP server must be available before uWSGI starts. For a more raw approach (working in unconnected mode) see the section on socket logging.

To enable conencted UDP mode pass the address of an UDP server to the `--daemonize`/`--logto` option:

```
./uwsgi -s :3031 -w simple_app --daemonize 192.168.0.100:1717
./uwsgi -s :3031 -w simple_app --logto 192.168.0.100:1717
```

This will redirect all the stdout/stderr data to the UDP socket on 192.168.0.100, port 1717. Now you need an UDP server that will manage your UDP messages. You could use netcat, or even uWSGI:

```
nc -u -p 1717 -s 192.168.0.100 -l
./uwsgi --udp 192.168.0.100:1717
```

The second way is a bit more useful as it will print the source (ip:port) of every message. In case of multiple uWSGI server logging on the same UDP server it will allow you to recognize one server from another. Naturally you can write your own apps to manage/filter/save the logs received via udp.

3.25.2 Pluggable loggers

uWSGI also supports pluggable loggers, which allow you more flexibility on where and what to log. Depending on the configuration of your uWSGI build, some loggers may or may not be available. Some may require to be loaded as plugins. To find out what plugins are available in your build, invoke uWSGI with `--logger-list`. To set up a pluggable logger, use the `--logger` or `--req-logger` options. `--logger` will set up a logger for every message while `--req-logger` will set up a logger for request information messages.

This is the syntax:

```
--logger <plugin>[:options]"
--logger "<name> <plugin>[:options]" # The quotes are only required on the command line -- config fi
```

You may set up as many loggers as you like. Named plugins are used for log routing. A very simple example of split request/error logging using plain text files follows.

```
[uwsgi]
req-logger = file:/tmp/reqlog
logger = file:/tmp/errlog
```

3.25.3 Log routing

By default all log lines are sent to all declared loggers. If this is not what you want, you can use `--log-route` (and `--log-req-route` for request loggers) to specify a regular expression to route certain log messages to different destinations.

For instance:

```
[uwsgi]
logger = mylogger1 syslog
logger = theredisone redislog:127.0.0.1:6269
logger = theredistwo redislog:127.0.0.1:6270
logger = file:/tmp/foobar # This logger will log everything as it's not named
logger = internalservererror file:/tmp/errors
# ...
log-route = internalservererror (HTTP/1.\d 500)
log-route = mylogger1 uWSGI listen queue of socket .* full
```

This will log each 500 level error to /tmp/errors, while listen queue full errors will end up in /tmp/foobar. This is somewhat similar to the *The uWSGI alarm subsystem (from 1.3)*, though alarms are usually heavier and should only be used for critical situations.

3.25.4 Logging to files

`logfile` plugin – embedded by default.

3.25.5 Logging to sockets

`logsocket` plugin – embedded by default.

You can log to an unconnected UNIX or UDP socket using `--logger socket:...` (or `--log-socket ...`).

```
uwsgi --socket :3031 --logger socket:/tmp/uwsgi.logsock
```

will send log entries to the Unix socket `/tmp/uwsgi.logsock`.

```
uwsgi --socket :3031 --logger socket:192.168.173.19:5050
```

will send log datagrams to the UDP address 192.168.173.19 on port 5050. You may also multicast logs to multiple log servers by passing the multicast address:

```
uwsgi --socket :3031 --logger socket:225.1.1.1:1717
```

3.25.6 Logging to syslog

`logsyslog` plugin – embedded by default

The `logsyslog` plugin routes logs to Unix standard syslog. You may pass an optional ID to send and the "facility" for the log entry.

```
uwsgi --socket :3031 --logger syslog:uwsgi1234
```

or

```
uwsgi --socket :3031 --logger syslog:uwsgi1234,local6
```

to send to the local6 facility

3.25.7 Logging to remote syslog

`logrsyslog` plugin – embedded by default

The `logrsyslog` plugin routes logs to Unix standard syslog residing on a remote server. In addtition to the address+port of the remote syslog server, you may pass an optional ID to send as the "facility" parameter for the log entry.

```
uwsgi --socket :3031 --logger rsyslog:12.34.56.78:12345,uwsgi1234
```

3.25.8 Redis logger

`redislog` plugin – embedded by default.

By default the `redislog` plugin will 'publish' each logline to a redis pub/sub queue. The logger plugin syntax is:

```
--logger redislog[:<host>,<command>,<prefix>]
```

By default `host` is mapped to `127.0.0.1:6379`, `command` is mapped to "publish uwsgi" and `prefix` is empty. To publish to a queue called foobar, use `redislog:127.0.0.1:6379,publish foobar`. Redis logging is not limited to pub/sub. You could for instance push items into a list, as in the next example.

```
--logger redislog:/tmp/redis.sock,rpush foo,example.com
```

As error situations could cause the master to block while writing a log line to a remote server, it's a good idea to use `--threaded-logger` to offload log writes to a secondary thread.

3.25.9 MongoDB logger

`mongodblog` plugin – embedded by default.

The logger syntax for MongoDB logging (`mongodblog`) is

```
--logger mongodblog[:<host>,<collection>,<node>]
```

Where `host` is the address of the MongoDB instance (default `127.0.0.1:27017`), `collection` names the collection to write log lines into (default `uwsgi.logs`) and `node` is an identification string for the instance sending logs (default: server hostname).

```
--logger mongodblog
```

Will run the logger with default values, while

```
--logger mongodblog:127.0.0.1:9090,foo.bar
```

Will write logs to the mongodb server 127.0.0.1:9090 in the collection `foo.bar` using the default node name. As with the Redis logger, offloading log writes to a dedicated thread is a good choice.

```
[uwsgi]
threaded-logger = true
logger = mongodblog:127.0.0.1:27017,uwsgi.logs_of_foobar
# As usual, you could have multiple loggers:
# logger = mongodblog:192.168.173.22:27017,uwsgi.logs_of_foobar
socket = :3031
```

3.25.10 ZeroMQ logging

As with UDP logging you can centralize/distribute logging via ZeroMQ. Build your logger daemon using a `ZMQ_PULL` socket:

```
import zmq

ctx = zmq.Context()

puller = ctx.socket(zmq.PULL)
puller.bind("tcp://192.168.173.18:9191")

while True:
    message = puller.recv()
    print message,
```

Now run your uWSGI server:

```
uwsgi --logger zeromq:tcp://192.168.173.18:9191 --socket :3031 --module werkzeug.testapp:test_app
```

(`--log-zeromq` is an alias for this logger.)

3.25.11 Crypto logger (plugin)

If you host your applications on cloud services without persistent storage you may want to send your logs to external systems. However logs often contain sensitive information that should not be transferred in clear. The `logcrypto` plugin logger attempts to solve this issue by encrypting each log packet before sending it over UDP to a server able to decrypt it. The next example will send each log packet to a UDP server available at 192.168.173.22:1717 encrypting the text with the secret key `ciaociao` with Blowfish in CBC mode.

```
uwsgi --plugin logcrypto --logger crypto:addr=192.168.173.22:1717,algo=bf-cbc,secret=ciaociao -M -p 4
```

An example server is available at https://github.com/unbit/uwsgi/blob/master/contrib/cryptologger.rb

3.25.12 Graylog2 logger (plugin)

`graylog2` plugin – not compiled by default.

This plugin will send logs to a Graylog2 server in Graylog2's native GELF format.

```
uwsgi --plugin graylog2 --logger graylog2:127.0.0.1:1234,dsfargeg
```

3.25.13 Systemd logger (plugin)

`systemd_logger` plugin – not compiled by default.

This plugin will write log entries into the Systemd journal.

```
uwsgi --plugin systemd_logger --logger systemd
```

3.25.14 Writing your own logger plugins

This plugin, `foolog.c` will write your messages in the file specified with –logto/–daemonize with a simple prefix using vector IO.

```
#include <uwsgi.h>

ssize_t uwsgi_foolog_logger(struct uwsgi_logger *ul, char *message, size_t len) {

        struct iovec iov[2];

        iov[0].iov_base = "[foo] ";
        iov[0].iov_len = 6;

        iov[1].iov_base = message;
        iov[1].iov_len = len;

        return writev(uwsgi.original_log_fd, iov, 2);
}

void uwsgi_foolog_register() {
        uwsgi_register_logger("syslog", uwsgi_syslog_logger);
}

struct uwsgi_plugin foolog_plugin = {
    .name = "foolog",
```

```
    .on_load = uwsgi_foolog_register,
};
```

3.26 Formatting uWSGI requests logs

uWSGI has a `--logformat` option for building custom request loglines. The syntax is simple:

```
[uwsgi]
logformat = i am a logline reporting "%(method) %(uri) %(proto)" returning with status %(status)
```

All of the variables marked with `%()` are substituted using specific rules. Three kinds of logvars are defined ("offsetof", functions, and user-defined).

3.26.1 offsetof

These are taken blindly from the internal `wsgi_request` structure of the current request.

- `%(uri)` -> REQUEST_URI
- `%(method)` -> REQUEST_METHOD
- `%(user)` -> REMOTE_USER
- `%(addr)` -> REMOTE_ADDR
- `%(host)` -> HTTP_HOST
- `%(proto)` -> SERVER_PROTOCOL
- `%(uagent)` -> HTTP_USER_AGENT (starting from 1.4.5)
- `%(referer)` -> HTTP_REFERER (starting from 1.4.5)

3.26.2 functions

These are simple functions called for generating the logvar value:

- `%(status)` -> HTTP response status code
- `%(micros)` -> response time in microseconds
- `%(msecs)` -> response time in milliseconds
- `%(time)` -> timestamp of the start of the request
- `%(ctime)` -> ctime of the start of the request
- `%(epoch)` -> the current time in Unix format
- `%(size)` -> response body size + response headers size (since 1.4.5)
- `%(ltime) -> human-formatted (Apache style)` request time (since 1.4.5)
- `%(hsize)` -> response headers size (since 1.4.5)
- `%(rsize)` -> response body size (since 1.4.5)
- `%(cl)` -> request content body size (since 1.4.5)
- `%(pid)` -> pid of the worker handling the request (since 1.4.6)

- `%(wid)` -> id of the worker handling the request (since 1.4.6)
- `%(switches)` -> number of async switches (since 1.4.6)
- `%(vars)` -> number of CGI vars in the request (since 1.4.6)
- `%(headers)` -> number of generated response headers (since 1.4.6)
- `%(core)` -> the core running the request (since 1.4.6)
- `%(vsz)` -> address space/virtual memory usage (in bytes) (since 1.4.6)
- `%(rss)` -> RSS memory usage (in bytes) (since 1.4.6)
- `%(vszM)` -> address space/virtual memory usage (in megabytes) (since 1.4.6)
- `%(rssM)` -> RSS memory usage (in megabytes) (since 1.4.6)
- `%(pktsize)` -> size of the internal request uwsgi packet (since 1.4.6)
- `%(modifier1)` -> modifier1 of the request (since 1.4.6)
- `%(modifier2)` -> modifier2 of the request (since 1.4.6)
- `%(metric.XXX)` -> access the XXX metric value (see *The Metrics subsystem*)
- `%(rerr)` -> number of read errors for the request (since 1.9.21)
- `%(werr)` -> number of write errors for the request (since 1.9.21)
- `%(ioerr)` -> number of write and read errors for the request (since 1.9.21)
- `%(tmsecs)` -> timestamp of the start of the request in milliseconds since the epoch (since 1.9.21)
- `%(tmicros)` -> timestamp of the start of the request in microseconds since the epoch (since 1.9.21)
- `%(var.XXX)` -> the content of request variable XXX (like var.PATH_INFO, available from 1.9.21)

3.26.3 User-defined logvars

You can define logvars within your request handler. These variables live only per-request.

```
import uwsgi
def application(env, start_response):
    uwsgi.set_logvar('foo', 'bar')
    # returns 'bar'
    print uwsgi.get_logvar('foo')
    uwsgi.set_logvar('worker_id', str(uwsgi.worker_id()))
    ...
```

With the following log format you will be able to access code-defined logvars:

```
uwsgi --logformat 'worker id = %(worker_id) for request "%(method) %(uri) %(proto)" test = %(foo)'
```

3.26.4 Apache-style combined request logging

To generate Apache-compatible logs:

```
[uwsgi]
...
log-format = %(addr) - %(user) [%(ltime)] "%(method) %(uri) %(proto)" %(status) %(size) "%(referer)"
...
```

3.26.5 Hacking logformat

(Updated to 1.9.21)

You can register new "logchunk" (the function to call for each logformat symbol) with

```
struct uwsgi_logchunk *uwsgi_register_logchunk(char *name, ssize_t (*func)(struct wsgi_request *, cha
```

`name` is the name of the symbol

`need_free` if 1, means the pointer set by `func` must be free()d;

`func` the function to call in the log handler

```
static ssize_t uwsgi_lf_foobar(struct wsgi_request *wsgi_req, char **buf) {
        *buf = uwsgi_num2str(wsgi_req->status);
        return strlen(*buf);
}

static void register_logchunks() {
        uwsgi_register_logchunk("foobar", uwsgi_lf_foobar, 1);
}

struct uwsgi_plugin foobar_plugin = {
        .name = "foobar",
        .on_load = register_logchunks,
};
```

if you load the foobar plugin, you will be able to use the %(foobar) request logging variable (reporting the request status)

3.27 Log encoders

uWSGI 1.9.16 got the "log encoding" feature.

An encoder receives a logline and give back a "transformation" of it.

Encoders can be added by plugins, and can be enabled in chain (the output of an encoder will be the input of the following one and so on).

```
[uwsgi]
; send logs to udp address 192.168.173.13:1717
logger = socket:192.168.173.13:1717
; before sending a logline to the logger encode it in gzip
log-encoder = gzip
; after gzip add a 'clear' prefix to easy decode
log-encoder = prefix i am gzip encoded
...
```

with this configuration the log server will receive the "i am gzip encoded" string followed by the tru log message encoded in gzip

The log encoder syntax is the following:

log-encoder = <encoder>[args]

so args (if any) are separated by a single space

3.27.1 Request logs VS stdout/stderr

The –log-encoder option encodes only the stdout/stderr logs.

If you want to encode request logs to use –log-req-encoder:

```
[uwsgi]
; send request logs to udp address 192.168.173.13:1717
req-logger = socket:192.168.173.13:1717
; before sending a logline to the logger encode it in gzip
log-req-encoder = gzip
; after gzip add a 'clear' prefix to easy decode
log-req-encoder = prefix i am gzip encoded
...
```

3.27.2 Routing encoders

Log routing allows sending each logline to a different log engine based on regexps. You can use the same system with encoders too:

```
[uwsgi]
; by default send logs to udp address 192.168.173.13:1717
logger = socket:192.168.173.13:1717
; an alternative logger using the same address
logger = secondlogger socket:192.168.173.13:1717
; use 'secondlogger' for the logline containing 'uWSGI'
log-route = secondlogger uWSGI
; before sending a logline to the 'secondlogger' logger encode it in gzip
log-encoder = gzip:secondlogger
...
```

3.27.3 `Core` encoders

The following encoders are available in the uwsgi 'core':

`prefix` add a raw prefix to each log msg

`suffix` add a raw suffix to each log msg

`nl` add a newline char to each log msg

`gzip` compress each msg with gzip (requires zlib)

`compress` compress each msg with zlib compress (requires zlib)

`format` apply the specified format to each log msg:

```
[uwsgi]
...
log-encoder = format [FOO ${msg} BAR]
...
```

`json` like `format` but each variable is json escaped

```
[uwsgi]
...
log-encoder = json {"unix":${unix}, "msg":"${msg}"}
...
```

The following variables (for format and json) are available:

`${msg}` the raw log message (newline stripped)

`${msgnl}` the raw log message (with newline)

`${unix}` the current unix time

`${micros}` the current unix time in microseconds

`${strftime:xxx}` strftime using the xxx format:

```
[uwsgi]
...
; we need to escape % to avoid magic vars nameclash
log-encoder = json {"unix":${unix}, "msg":"${msg}", "date":"${strftime:%%d/%%m/%%Y %%H:%%M:%%S}"}
...
```

3.27.4 The `msgpack` encoder

This is the first log-encoder plugin officially added to uWSGI sources. It allows encoding of loglines in msgpack (http://msgpack.org/) format.

The syntax is pretty versatile as it has been developed for adding any information to a single packet

`log-encoder = msgpack <format>`

format is pretty complex as it is a list of the single items in the whole packet.

For example if you want to encode the {'foo':'bar', 'test':17} dictionary you need to read it as:

a map of 2 items | the string foo | the string bar | the string test | the integer 17

for a total of 5 items.

A more complex structure {'boo':30, 'foo':'bar', 'test': [1,3,3,17.30,nil,true,false]}

will be

a map of 3 items | the string boo | the number 30| the string foo| the string bar | the string test | an array of 7 items | the integer 1 | the integer 3 | the integer 3 | the float 17.30 | a nil | a true | a false

The <format> string is a representation of this way:

`map:2|str:foo|str:bar|str:test|int:17`

The pipe is the seprator of each item. The string before the colon is the type of item, followed by the optional argument

The following item types are supported:

`map` a dictionary, the argument is the number of items

`array` an array, the argument is the number of items

`str` a string, the argument is the string itself

`bin` a byte array, the argument is the binary stream itself

`int` an integer, the argument is the number

`float` a float, the argument is the number

`nil` undefined/NULL

`true` boolean TRUE

`false` boolean FALSE

in addition to msgpack types, a series of dynamic types are available:

`msg` translate the logline to a msgpack string with newline chopped

`msgbin` translate the logline to a msgpack byte array with newline chopped

`msgnl` translate the logline to a msgpack string (newline included)

`msgbin` translate the logline to a msgpack byte array (newline included)

`unix` translate to an integer of the unix time

`micros` translate to an integer of the unix time in microseconds

`strftime` translate to a string using strftime syntax. The strftime format is the argument

As an example you can send logline to a logstash server via udp:

(logstash debug configuration):

```
input {
     udp {
              codec =>   msgpack {}
              port => 1717
     }
}
output {
     stdout { debug => true }
     elasticsearch { embedded => true }
}
```

```
[uwsgi]
logger = socket:192.168.173.13:1717
log-encoder = msgpack map:4|str:message|msg|str:hostname|str:%h|str:version|str:%V|str:appname|str:my
...
```

this will generate the following structure:

```
{
   "message": "*** Starting uWSGI 1.9.16-dev-29d80ce (64bit) on [Sat Sep  7 15:04:32 2013] ***",
   "hostname": "unbit.it",
   "version": "1.9.16-dev",
   "appname": "myapp"
}
```

that will be stored in elasticsearch

3.27.5 Notes

Encoders automatically enable –log-master

For best performance consider allocating a thread for log sending with –threaded-logger

3.28 Hooks

(updated to uWSGI 1.9.16)

uWSGI main directive is being “modular”. The vast majority of its features are exposed as plugins, both to allow users to optimize their build and to encourage developers to extend it.

Writing plugins can be an annoying task, expecially if you only need to change/implement a single function.

For simple tasks, uWSGI exposes an hook api you can abuse to modify uWSGI internal behaviours.

3.28.1 The uWSGI "hookable" phases

Before being ready to manage requests, uWSGI go through various "phases". You can attach one or more "hooks" to that phases.

Each phase can be "fatal", if so, a failing hook will mean failing of the whole uWSGI instance (generally calling exit(1))

Currently (September 2013) the following phases are available:

`asap` run directly after configuration file has been parsed, before anything else is done. it is fatal.

`pre-jail` run before any attempt to drop privileges or put the process in some form of jail. it is fatal.

`post-jail` run soon after any jailing, but before privileges drop. If jailing requires fork(), the parent process run this phase. it is fatal.

`in-jail` run soon after jayling, but after post-jail. If jailing requires fork(), the chidlren run this phase. it is fatal.

`as-root` run soon before privileges drop (last chance to run something as root). it is fatal.

`as-user` run soon after privileges drop. it is fatal.

`pre-app` run before applications loading. it is fatal.

`post-app` run after applications loading. it is fatal.

`accepting` run before the each worker starts accepting requests (available from uWSGI 1.9.21).

`accepting1` run before the first worker starts accepting requests (available from uWSGI 1.9.21).

`accepting-once` run before the each worker starts accepting requests (available from uWSGI 1.9.21, runs one time per instance).

`accepting1-once` run before the first worker starts accepting requests (available from uWSGI 1.9.21, runs one time per instance).

`as-user-atexit` run before shutdown of the instance. it is non-fatal.

`as-emperor` run soon after the spawn of a vassal in the Emperor process. it is non-fatal.

`as-vassal` run in the vassal before executing the uwsgi binary. it is fatal.

3.28.2 The "hardcoded" hooks

As said before the purpose of the hook subsystem is allowing to attach "hooks" to the various uWSGI phases.

There are two kind of hooks, the simple ones are the so-called "hardcoded". They exposes common patterns at the cost of versatility.

Currently (Semptember 2013) the following "hardcoded" hooks are available (they run in the order they are showed below):

`mount` mount filesystems

arguments: <filesystem> <src> <mountpoint> [flags]

the exposed flags are the one available for the operating system. As an example on linux you will options like bind, recursive, readonly and so on

`umount` un-mount filesystems

arguments: <mountpoint> [flags]

`exec` run shell commands

arguments: <command> [args...]

run the command under /bin/sh.

If for some reason you do not want to use /bin/sh as the running shell, you can override it with the –binsh option (you can specify multiple –binsh option, they will be tried until one valid shell is found)

`call` call functions in the current process address space

arguments: <symbol> [args...]

generally the arguments are ignored (the only exceptions are the emperor/vassal phases, see below) as the system expect to call symbol without arguments.

<symbol> can be any symbol currently available in the process address space.

This allows funny tricks, abusing the –dlopen uWSGI option:

```
// foo.c
#include <stdio.h>
void foo_hello() {
        printf("i am the foo_hello function called by a hook !!!\n");
}
```

build it as shared library:

```
gcc -o foo.so -shared -fPIC foo.c
```

and load into the uWSGI symbols table:

```
uwsgi --dlopen ./foo.so ...
```

from now on the “foo_hello” symbol is available in the uWSGI symbols table, ready to be called by the ‘call’ hooks

Pay attention as –dlopen is a wrapper for the C function dlopen(), so beware of absolute paths and library search paths (if you do not want headaches, use always absolute paths when dealing with shared libraries)

3.28.3 Attaching “hardcoded” hooks

Each hardcoded hooks exposes is set of options for each phase (with some exception)

Each option is composed by the name of the hook and its phase, so to run a command in the ‘as-root’ phase you will use –exec-as-root, or –exec-as-user for the ‘as-user’ phase.

Remember, you can attach all of the hooks you need to a hook-phase pair:

```
[uwsgi]
...
exec-as-root = cat /proc/cpuinfo
exec-as-root = echo 1 > /proc/sys/net/ipv4/ip_forward

exec-as-user = ls /tmp
exec-as-user-at-exit = rm /tmp/foobar

dlopen = ./foo.so
call-as-user = foo_hello
...
```

The only exception to the rule are the *as-emperor* and *as-vassal* phases. For various reasons they expose a bunch of handy variants

3.28.4 The "advanced" hooks

A problem (limiting their versatility) with 'hardcoded' hooks, is that you cannot control the order of the whole chain (as each phase executes each hooks grouped by type). If you want more control "advanced" hooks are the best choice.

Each phase has a single chain in which you specify the hook the call and which handler.

Handlers specify how to run hooks. New handlers can be registered by plugins.

Currently the handlers exposed by the core are:

`exec` same as the 'exec' hardcoded options

`call` call the specified symbol ignoring return value

`callret` call the specified symbol expecting an int return. anything != 0 means failure

`callint` call the specified symbol parsing the argument as an int

`callintret` call the specified symbol parsing the argument as an int and expecting an int return.

`mount` same as 'mount' hardcoded options

`umount` same as 'umount' hardcoded options

`cd` commodity handler, you can obtain the same using callint:chdir <directory>

`exit` commodity handler, you can obtain the same using callint:exit [num]

`print` commodity handler, you can obtain the same calling the uwsgi_log symbol

`write` (from uWSGI 1.9.21), write a string to the specified file using write:<file> <string>

`writefifo` (from uWSGI 1.9.21), write a string to the specified fifo using writefifo:<file> <string>

`unlink` (from uWSGI 1.9.21), unlink the specified file

```
[uwsgi]
...
hook-as-root = mount:proc none /proc
hook-as-root = exec:cat /proc/self/mounts
hook-pre-app = callint:putenv PATH=bin:$(PATH)
hook-post-app = call:uwsgi_log application has been loaded
hook-as-user-atexit = print:goodbye cruel world
...
```

3.29 Glossary

harakiri A feature of uWSGI that aborts workers that are serving requests for an excessively long time. Configured using the `harakiri` family of options. Every request that will take longer than the seconds specified in the harakiri timeout will be dropped and the corresponding worker recycled.

master uWSGI's built-in prefork+threading multi-worker management mode, activated by flicking the `master` switch on. For all practical serving deployments it's not really a good idea *not* to use master mode.

3.30 uWSGI third party plugins

The following plugin (unless differently specified) are not commercially supported.

Feel free to add your plugin to the list sending a pull request

3.30.1 uwsgi-capture

License: MIT

Author: unbit

allows gathering video4linux frames in a sharedarea

Website: https://github.com/unbit/uwsgi-capture

3.30.2 uwsgi-wstcp

License: MIT

Author: unbit

maps websockets to tcp connections (useful for proxying via javascript)

Website: https://github.com/unbit/uwsgi-wstcp

3.30.3 uwsgi-pgnotify

License: MIT

Author: unbit

integrate postgresql notification system with uWSGI signal framework

Website: https://github.com/unbit/uwsgi-pgnotify

3.30.4 uwsgi-quota

License: MIT

Author: unbit

allows to set and monitor filesystem quota

Website: https://github.com/unbit/uwsgi-quota

3.30.5 uwsgi-eventfd

License: MIT

Author: unbit

allows to monitor eventfd() objects (like events sent by the cgroup system)

Website: https://github.com/unbit/uwsgi-eventfd

3.30.6 uwsgi-console-broadcast

License: MIT

Author: unbit

exposes hooks for sending broadcast messages to users terminals

Website: https://github.com/unbit/uwsgi-console-broadcast

3.30.7 uwsgi-strophe

License: MIT

Author: unbit

integration with the libstrophe library (xmpp)

Website: https://github.com/unbit/uwsgi-strophe

3.30.8 uwsgi-alarm-chain

License: MIT

Author: unbit

virtual alarm handler combining multiple alarms in a single one

Website: https://github.com/unbit/uwsgi-alarm-chain

3.30.9 uwsgi-netlink

License: MIT

Author: unbit

integration with the linux netlink subsystem

Website: https://github.com/unbit/uwsgi-netlink

3.30.10 uwsgi-pushover

License: MIT

Author: unbit

integration with the pushover.net services

Website: https://github.com/unbit/uwsgi-pushover

CHAPTER 4

Tutorials

4.1 The uWSGI Caching Cookbook

This is a cookbook of various caching techniques using *uWSGI internal routing*, *The uWSGI caching framework* and *uWSGI Transformations*

The examples assume a modular uWSGI build. You can ignore the 'plugins' option, if you are using a monolithic build.

Recipes are tested over uWSGI 1.9.7. Older versions may not work.

4.1.1 Let'start

This is a simple perl/PSGI Dancer app we deploy on an http-socket with 4 processes

```
use Dancer;

get '/' => sub {
        "Hello World!"
};

dance;
```

This is the uWSGI config, pay attention to the log-micros directive. The objective of uWSGI in-memory caching is generating a response in less than 1 millisecond (yes, this is true), so we want to get the response time logging in microseconds.

```
[uwsgi]
; load the PSGI plugin as the default one
plugins = 0:psgi
; load the Dancer app
psgi = myapp.pl
; enable the master process
master = true
; spawn 4 processes
processes = 4
; bind an http socket to port 9090
http-socket = :9090
; log response time with microseconds resolution
log-micros = true
```

Run the uWSGI instance in your terminal and just make a bunch of requests to it

```
curl -D /dev/stdout http://localhost:9090/
```

If all goes well you should see something similar in your uWSGI logs:

```
[pid: 26586|app: 0|req: 1/1] 192.168.173.14 () {24 vars in 327 bytes} [Wed Apr 17 09:06:58 2013] GET
[pid: 26586|app: 0|req: 2/2] 192.168.173.14 () {24 vars in 327 bytes} [Wed Apr 17 09:07:14 2013] GET
[pid: 26586|app: 0|req: 3/3] 192.168.173.14 () {24 vars in 327 bytes} [Wed Apr 17 09:07:16 2013] GET
[pid: 26586|app: 0|req: 4/4] 192.168.173.14 () {24 vars in 327 bytes} [Wed Apr 17 09:07:17 2013] GET
[pid: 26586|app: 0|req: 5/5] 192.168.173.14 () {24 vars in 327 bytes} [Wed Apr 17 09:07:18 2013] GET
```

while curl will return:

```
HTTP/1.1 200 OK
Server: Perl Dancer 1.3112
Content-Length: 12
Content-Type: text/html
X-Powered-By: Perl Dancer 1.3112

Hello World!
```

The first request on a process took about 3 milliseconds (this is normal as lot of code is executed the first time), but the following run in about 1 millisecond).

Now we want to store the response in the uWSGI cache.

4.1.2 The first recipe

We first create a uWSGI cache named 'mycache' with 100 slot of 64k (new options are at the end of the config) and at each request for '/' we search in it for a specific item named 'myhome'.

This time we load the router_cache plugin too (it is builtin by default in monolithic servers)

```
[uwsgi]
; load the PSGI plugin as the default one
plugins = 0:psgi,router_cache
; load the Dancer app
psgi = myapp.pl
; enable the master process
master = true
; spawn 4 processes
processes = 4
; bind an http socket to port 9090
http-socket = :9090
; log response time with microseconds resolution
log-micros = true

; create a cache with 100 items (default size per-item is 64k)
cache2 = name=mycache,items=100
; at each request for / check for a 'myhome' item in the 'mycache' cache
; 'route' apply a regexp to the PATH_INFO request var
route = ^/$ cache:key=myhome,name=mycache
```

restart uWSGI and re-run the previous test with curl. Sadly nothing will change. Why ?

Because you did not instructed uWSGI to store the plugin response in the cache. You need to use the cachestore routing action

```
[uwsgi]
; load the PSGI plugin as the default one
```

```
plugins = 0:psgi,router_cache
; load the Dancer app
psgi = myapp.pl
; enable the master process
master = true
; spawn 4 processes
processes = 4
; bind an http socket to port 9090
http-socket = :9090
; log response time with microseconds resolution
log-micros = true

; create a cache with 100 items (default size per-item is 64k)
cache2 = name=mycache,items=100
; at each request for / check for a 'myhome' item in the 'mycache' cache
; 'route' apply a regexp to the PATH_INFO request var
route = ^/$ cache:key=myhome,name=mycache
; store each successfull request (200 http status code) for '/' in the 'myhome' item
route = ^/$ cachestore:key=myhome,name=mycache
```

Now re-run the test, and you should see requests going down to a range of 100-300 microseconds (it depends on various factors, but you should gain at least 60% in response time)

Log line report -1 as the app id:

```
[pid: 26703|app: -1|req: -1/2] 192.168.173.14 () {24 vars in 327 bytes} [Wed Apr 17 09:24:52 2013] GE
```

this is because when a response is served from the cache your app/plugin is not touched (in this case, no perl call is involved)

You will note less headers too:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 12

Hello World!
```

This is because only the body of a response is cached. By default the generated response is set as text/html but you can change it or let the mime types engine do the work for you (see later)

4.1.3 Cache them all !!!

We want to cache all of our requests. Some of them returns images and css, while the others are always text/html

```
[uwsgi]
; load the PSGI plugin as the default one
plugins = 0:psgi,router_cache
; load the Dancer app
psgi = myapp.pl
; enable the master process
master = true
; spawn 4 processes
processes = 4
; bind an http socket to port 9090
http-socket = :9090
; log response time with microseconds resolution
log-micros = true
```

```
; create a cache with 100 items (default size per-item is 64k)
cache2 = name=mycache,items=100
; load the mime types engine
mime-file = /etc/mime.types

; at each request starting with /img check it in the cache (use mime types engine for the content typ
route = ^/img/(.+) cache:key=/img/$1,name=mycache,mime=1

; at each request ending with .css check it in the cache
route = \.css$ cache:key=${REQUEST_URI},name=mycache,content_type=text/css

; fallback to text/html all of the others request
route = .* cache:key=${REQUEST_URI},name=mycache
; store each successfull request (200 http status code) in the 'mycache' cache using the REQUEST_URI
route = .* cachestore:key=${REQUEST_URI},name=mycache
```

4.1.4 Multiple caches

You may want/need to store items in different caches. We can chnage the previous recipe to use three different caches for images, css and html responses.

```
[uwsgi]
; load the PSGI plugin as the default one
plugins = 0:psgi,router_cache
; load the Dancer app
psgi = myapp.pl
; enable the master process
master = true
; spawn 4 processes
processes = 4
; bind an http socket to port 9090
http-socket = :9090
; log response time with microseconds resolution
log-micros = true

; create a cache with 100 items (default size per-item is 64k)
cache2 = name=mycache,items=100

; create a cache for images with dynamic size (images can be big, so do not waste memory)
cache2 = name=images,items=20,bitmap=1,blocks=100

; a cache for css (20k per-item is more than enough)
cache2 = name=stylesheets,items=30,blocksize=20000

; load the mime types engine
mime-file = /etc/mime.types

; at each request starting with /img check it in the 'images' cache (use mime types engine for the co
route = ^/img/(.+) cache:key=/img/$1,name=images,mime=1

; at each request ending with .css check it in the 'stylesheets' cache
route = \.css$ cache:key=${REQUEST_URI},name=stylesheets,content_type=text/css

; fallback to text/html all of the others request
route = .* cache:key=${REQUEST_URI},name=mycache
```

```
; store each successfull request (200 http status code) in the 'mycache' cache using the REQUEST_URI
route = .* cachestore:key=${REQUEST_URI},name=mycache
; store images and stylesheets in the corresponding caches
route = ^/img/ cachestore:key=${REQUEST_URI},name=images
route = ^/css/ cachestore:key=${REQUEST_URI},name=stylesheets
```

Important, every matched 'cachestore' will overwrite the previous one. So we are putting .* as the first rule.

4.1.5 Being more aggressive, the Expires HTTP header

You can set an expiration for each cache item. If an item has an expire, it will be translated to an HTTP Expires headers. This means, once you have sent a cache item to the browser, it will not request it until it expires !!!

We use the previous recipe simply adding different expires to the items

```
[uwsgi]
; load the PSGI plugin as the default one
plugins = 0:psgi,router_cache
; load the Dancer app
psgi = myapp.pl
; enable the master process
master = true
; spawn 4 processes
processes = 4
; bind an http socket to port 9090
http-socket = :9090
; log response time with microseconds resolution
log-micros = true

; create a cache with 100 items (default size per-item is 64k)
cache2 = name=mycache,items=100

; create a cache for images with dynamic size (images can be big, so do not waste memory)
cache2 = name=images,items=20,bitmap=1,blocks=100

; a cache for css (20k per-item is more than enough)
cache2 = name=stylesheets,items=30,blocksize=20000

; load the mime types engine
mime-file = /etc/mime.types

; at each request starting with /img check it in the 'images' cache (use mime types engine for the co
route = ^/img/(.+) cache:key=/img/$1,name=images,mime=1

; at each request ending with .css check it in the 'stylesheets' cache
route = \.css$ cache:key=${REQUEST_URI},name=stylesheets,content_type=text/css

; fallback to text/html all of the others request
route = .* cache:key=${REQUEST_URI},name=mycache

; store each successfull request (200 http status code) in the 'mycache' cache using the REQUEST_URI
route = .* cachestore:key=${REQUEST_URI},name=mycache,expires=60
; store images and stylesheets in the corresponding caches
route = ^/img/ cachestore:key=${REQUEST_URI},name=images,expires=3600
route = ^/css/ cachestore:key=${REQUEST_URI},name=stylesheets,expires=3600
```

images and stylesheets are cached for 1 hour, while html response are cached for 1 minute

4.1.6 Storing GZIP variant of an object

Back to the first recipe. We may want to store two copies of a response. The "clean" one and a gzipped one for clients supporting gzip encoding.

To enable the gzip copy you only need to choose a name for the item and pass it as the 'gzip' option of the cachestore action.

Then check for HTTP_ACCEPT_ENCODING request header. If it contains the 'gzip' word you can send it the gzip variant.

```
[uwsgi]
; load the PSGI plugin as the default one
plugins = 0:psgi,router_cache
; load the Dancer app
psgi = myapp.pl
; enable the master process
master = true
; spawn 4 processes
processes = 4
; bind an http socket to port 9090
http-socket = :9090
; log response time with microseconds resolution
log-micros = true

; create a cache with 100 items (default size per-item is 64k)
cache2 = name=mycache,items=100
; if the client support GZIP give it the gzip body
route-if = contains:${HTTP_ACCEPT_ENCODING};gzip cache:key=gzipped_myhome,name=mycache,content_encodi
; else give it the clear version
route = ^/$ cache:key=myhome,name=mycache

; store each successfull request (200 http status code) for '/' in the 'myhome' item in gzip too
route = ^/$ cachestore:key=myhome,gzip=gzipped_myhome,name=mycache
```

4.1.7 Storing static files in the cache for fast serving

You can populate a uWSGI cache on server startup with static files for fast serving them. The option –load-file-in-cache is the right tool for the job

```
[uwsgi]
plugins = 0:notfound,router_cache
http-socket = :9090
cache2 = name=files,bitmap=1,items=1000,blocksize=10000,blocks=2000
load-file-in-cache = files /usr/share/doc/socat/index.html
route-run = cache:key=${REQUEST_URI},name=files
```

You can specify all of the –load-file-in-cache directive you need but a better approach would be

```
[uwsgi]
plugins = router_cache
http-socket = :9090
cache2 = name=files,bitmap=1,items=1000,blocksize=10000,blocks=2000
for-glob = /usr/share/doc/socat/*.html
   load-file-in-cache = files %(_)
endfor =
route-run = cache:key=${REQUEST_URI},name=files
```

this will store all of the html files in /usr/share/doc/socat.

Items are stored with the path as the key.

When a non-existent item is requested the connection is closed and you should get an ugly

```
-- unavailable modifier requested: 0 --
```

This is because the internal routing system failed to manage the request, and no request plugin is available to manage the request.

You can build a better infrastructure using the simple 'notfound' plugin (it will always return a 404)

```
[uwsgi]
plugins = 0:notfound,router_cache
http-socket = :9090
cache2 = name=files,bitmap=1,items=1000,blocksize=10000,blocks=2000
for-glob = /usr/share/doc/socat/*.html
   load-file-in-cache = files %(_)
endfor =
route-run = cache:key=${REQUEST_URI},name=files
```

You can store file in the cache as gzip too using –load-file-in-cache-gzip

This option does not allow to set the name of the cache item, so to support client iwith and without gzip support we can use 2 different caches

```
[uwsgi]
plugins = 0:notfound,router_cache
http-socket = :9090
cache2 = name=files,bitmap=1,items=1000,blocksize=10000,blocks=2000
cache2 = name=compressedfiles,bitmap=1,items=1000,blocksize=10000,blocks=2000
for-glob = /usr/share/doc/socat/*.html
   load-file-in-cache = files %(_)
   load-file-in-cache-gzip = compressedfiles %(_)
endfor =
; take the item from the compressed cache
route-if = contains:${HTTP_ACCEPT_ENCODING};gzip cache:key=${REQUEST_URI},name=compressedfiles,conter
; fallback to the uncompressed one
route-run = cache:key=${REQUEST_URI},name=files
```

4.1.8 Caching for authenticated users

If you authenticate users with http basic auth, you can differentiate caching for each one using the ${REMOTE_USER} request variable:

```
[uwsgi]
; load the PSGI plugin as the default one
plugins = 0:psgi,router_cache
; load the Dancer app
psgi = myapp.pl
; enable the master process
master = true
; spawn 4 processes
processes = 4
; bind an http socket to port 9090
http-socket = :9090
; log response time with microseconds resolution
log-micros = true
```

```
; create a cache with 100 items (default size per-item is 64k)
cache2 = name=mycache,items=100
; check if the user is authenticated
route-if-not = empty:${REMOTE_USER} goto:cacheme
route-run = break:

; the following rules are executed only if REMOTE_USER is defined
route-label = cacheme
route = ^/$ cache:key=myhome_for_${REMOTE_USER},name=mycache
; store each successfull request (200 http status code) for '/'
route = ^/$ cachestore:key=myhome_for_${REMOTE_USER},name=mycache
```

Cookie-based authentication is generally more complex, but the vast majority of time a session id is passed as a cookie.

You may want to use this session_id as the key

```
[uwsgi]
; load the PHP plugin as the default one
plugins = 0:php,router_cache
; enable the master process
master = true
; spawn 4 processes
processes = 4
; bind an http socket to port 9090
http-socket = :9090
; log response time with microseconds resolution
log-micros = true

; create a cache with 100 items (default size per-item is 64k)
cache2 = name=mycache,items=100
; check if the user is authenticated
route-if-not = empty:${cookie[PHPSESSID]} goto:cacheme
route-run = break:

; the following rules are executed only if the PHPSESSID cookie is defined
route-label = cacheme
route = ^/$ cache:key=myhome_for_${cookie[PHPSESSID]},name=mycache
; store each successfull request (200 http status code) for '/'
route = ^/$ cachestore:key=myhome_for_${cookie[PHPSESSID]},name=mycache
```

Obviously a malicious user could build a fake session id and could potentially fill your cache. You should always check the session id. There is no single solution, but a good example for file-based php session is the following one:

```
[uwsgi]
; load the PHP plugin as the default one
plugins = 0:php,router_cache
; enable the master process
master = true
; spawn 4 processes
processes = 4
; bind an http socket to port 9090
http-socket = :9090
; log response time with microseconds resolution
log-micros = true

; create a cache with 100 items (default size per-item is 64k)
cache2 = name=mycache,items=100
; check if the user is authenticated
route-if-not = empty:${cookie[PHPSESSID]} goto:cacheme
```

```
route-run = break:

; the following rules are executed only if the PHPSESSID cookie is defined
route-label = cacheme
; stop if the session file does not exist
route-if-not = isfile:/var/lib/php5/sessions/sess_${cookie[PHPSESSID]} break:
route = ^/$ cache:key=myhome_for_${cookie[PHPSESSID]},name=mycache
; store each successfull request (200 http status code) for '/'
route = ^/$ cachestore:key=myhome_for_${cookie[PHPSESSID]},name=mycache
```

4.1.9 Caching to files

Sometimes, instead of caching in memory you want to store static files.

The transformation_tofile plugin allows you to store responses in files:

```
[uwsgi]
; load the PHP plugin as the default one
plugins = 0:psgi,transformation_tofile,router_static
; load the Dancer app
psgi = myapp.pl
; enable the master process
master = true
; spawn 4 processes
processes = 4
; bind an http socket to port 9090
http-socket = :9090
; log response time with microseconds resolution
log-micros = true

; check if a file exists
route-if = isfile:/var/www/cache/${hex[PATH_INFO]}.html static:/var/www/cache/${hex[PATH_INFO]}.html
; otherwise store the response in it
route-run = tofile:/var/www/cache/${hex[PATH_INFO]}.html
```

the hex[] routing var take a request variable content and encode it in hexadecimal. As PATH_INFO tend to contains / it is a better approach than storing full path names (or using other encoding scheme like base64 that can include slashes too)

4.2 Setting up Django and your web server with uWSGI and nginx

This tutorial is aimed at the Django user who wants to set up a production web server. It takes you through the steps required to set up Django so that it works nicely with uWSGI and nginx. It covers all three components, providing a complete stack of web application and server software.

Django Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.

nginx (pronounced *engine-x*) is a free, open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server.

4.2.1 Some notes about this tutorial

Note

This is a **tutorial**. It is not intended to provide a reference guide, never mind an exhaustive reference, to the subject of deployment.

---

nginx and uWSGI are good choices for Django deployment, but they are not the only ones, or the 'official' ones. There are excellent alternatives to both, and you are encouraged to investigate them.

The way we deploy Django here is a good way, but it is **not** the *only* way; for some purposes it is probably not even the best way.

It is however a reliable and easy way, and the material covered here will introduce you to concepts and procedures you will need to be familiar with whatever software you use for deploying Django. By providing you with a working setup, and rehearsing the steps you must take to get there, it will offer you a basis for exploring other ways to achieve this.

Note

This tutorial makes some assumptions about the system you are using.

It is assumed that you are using a Unix-like system, and that it features an aptitude-like package manager. However if you need to ask questions like "What's the equivalent of aptitude on Mac OS X?", you'll be able to find that kind of help fairly easily.

While this tutorial assumes Django 1.4 or later, which will automatically create a wsgi module in your new project, the instructions will work with earlier versions. You will though need to obtain that Django wsgi module yourself, and you may find that the Django project directory structure is slightly different.

4.2.2 Concept

A web server faces the outside world. It can serve files (HTML, images, CSS, etc) directly from the file system. However, it can't talk *directly* to Django applications; it needs something that will run the application, feed it requests from web clients (such as browsers) and return responses.

A Web Server Gateway Interface - WSGI - does this job. WSGI is a Python standard.

uWSGI is a WSGI implementation. In this tutorial we will set up uWSGI so that it creates a Unix socket, and serves responses to the web server via the WSGI protocol. At the end, our complete stack of components will look like this:

```
the web client <-> the web server <-> the socket <-> uwsgi <-> Django
```

4.2.3 Before you start setting up uWSGI

virtualenv

Make sure you are in a virtualenv for the software we need to install (we will describe how to install a system-wide uwsgi later):

```
virtualenv uwsgi-tutorial
cd uwsgi-tutorial
source bin/activate
```

Django

Install Django into your virtualenv, create a new project, and `cd` into the project:

```
pip install Django
django-admin.py startproject mysite
cd mysite
```

About the domain and port

In this tutorial we will call your domain `example.com`. Substitute your own FQDN or IP address.

Throughout, we'll be using port 8000 for the web server to publish on, just like the Django runserver does by default. You can use whatever port you want of course, but I have chosen this one so it doesn't conflict with anything a web server might be doing already.

4.2.4 Basic uWSGI installation and configuration

Install uWSGI into your virtualenv

```
pip install uwsgi
```

Of course there are other ways to install uWSGI, but this one is as good as any. Remember that you will need to have Python development packages installed. In the case of Debian, or Debian-derived systems such as Ubuntu, what you need to have installed is `pythonX.Y-dev`, where X.Y is your version of Python.

Basic test

Create a file called `test.py`:

```
# test.py
def application(env, start_response):
    start_response('200 OK', [('Content-Type','text/html')])
    return "Hello World"
```

Note: Take into account that Python 3 requires `bytes()`.

Run uWSGI:

```
uwsgi --http :8000 --wsgi-file test.py
```

The options mean:

- `http :8000`: use protocol http, port 8000
- `wsgi-file test.py`: load the specified file, test.py

This should serve a 'hello world' message directly to the browser on port 8000. Visit:

```
http://example.com:8000
```

to check. If so, it means the following stack of components works:

```
the web client <-> uWSGI <-> Python
```

Test your Django project

Now we want uWSGI to do the same thing, but to run a Django site instead of the `test.py` module.

First stop the running uwsgi process using Ctrl-C. Then make sure that your `mysite` project actually runs using Django's built-in web server:

```
python manage.py runserver 0.0.0.0:8000
```

If that works, stop Django's development server with Ctrl-C again, and run it using uWSGI:

```
uwsgi --http :8000 --module mysite.wsgi
```

- `module mysite.wsgi`: load the specified wsgi module

Point your browser at the server; if the site appears, it means uWSGI is able to serve your Django application from your virtualenv, and this stack operates correctly:

```
the web client <-> uWSGI <-> Django
```

Now normally we won't have the browser speaking directly to uWSGI. That's a job for the webserver, which will act as a go-between.

4.2.5 Basic nginx

Install nginx

```
sudo apt-get install nginx
sudo /etc/init.d/nginx start    # start nginx
```

And now check that the nginx is serving by visiting it in a web browser on port 80 - you should get a message from nginx: "Welcome to nginx!". That means these components of the full stack are working together:

```
the web client <-> the web server
```

If something else *is* already serving on port 80 and you want to use nginx there, you'll have to reconfigure nginx to serve on a different port. For this tutorial though, we're going to be using port 8000.

Configure nginx for your site

You will need the `uwsgi_params` file, which is available in the `nginx` directory of the uWSGI distribution, or from https://github.com/nginx/nginx/blob/master/conf/uwsgi_params

Copy it into your project directory. In a moment we will tell nginx to refer to it.

Now create a file called mysite_nginx.conf, and put this in it:

```
# mysite_nginx.conf

# the upstream component nginx needs to connect to
upstream django {
    # server unix:///path/to/your/mysite/mysite.sock; # for a file socket
    server 127.0.0.1:8001; # for a web port socket (we'll use this first)
}

# configuration of the server
server {
    # the port your site will be served on
```

```
    listen      8000;
    # the domain name it will serve for
    server_name .example.com; # substitute your machine's IP address or FQDN
    charset     utf-8;

    # max upload size
    client_max_body_size 75M;   # adjust to taste

    # Django media
    location /media  {
        alias /path/to/your/mysite/media;  # your Django project's media files - amend as required
    }

    location /static {
        alias /path/to/your/mysite/static; # your Django project's static files - amend as required
    }

    # Finally, send all non-media requests to the Django server.
    location / {
        uwsgi_pass  django;
        include     /path/to/your/mysite/uwsgi_params; # the uwsgi_params file you installed
    }
}
```

This conf file tells nginx to serve up media and static files from the filesystem, as well as handle requests that require Django's intervention. For a large deployment it is considered good practice to let one server handle static/media files, and another handle Django applications, but for now, this will do just fine.

Symlink to this file from /etc/nginx/sites-enabled so nginx can see it:

```
sudo ln -s ~/path/to/your/mysite/mysite_nginx.conf /etc/nginx/sites-enabled/
```

Deploying static files

Before running nginx, you have to collect all Django static files in the static folder. First of all you have to edit mysite/settings.py adding:

```
STATIC_ROOT = os.path.join(BASE_DIR, "static/")
```

and then run

```
python manage.py collectstatic
```

Basic nginx test

Restart nginx:

```
sudo /etc/init.d/nginx restart
```

To check that media files are being served correctly, add an image called `media.png` to the `/path/to/your/project/project/media directory`, then visit http://example.com:8000/media/media.png - if this works, you'll know at least that nginx is serving files correctly.

It is worth not just restarting nginx, but actually stopping and then starting it again, which will inform you if there is a problem, and where it is.

4.2.6 nginx and uWSGI and test.py

Let's get nginx to speak to the "hello world" `test.py` application.

```
uwsgi --socket :8001 --wsgi-file test.py
```

This is nearly the same as before, except this time one of the options is different:

- `socket :8001`: use protocol uwsgi, port 8001

nginx meanwhile has been configured to communicate with uWSGI on that port, and with the outside world on port 8000. Visit:

http://example.com:8000/

to check. And this is our stack:

```
the web client <-> the web server <-> the socket <-> uWSGI <-> Python
```

Meanwhile, you can try to have a look at the uswgi output at http://example.com:8001 - but quite probably, it won't work because your browser speaks http, not uWSGI, though you should see output from uWSGI in your terminal.

4.2.7 Using Unix sockets instead of ports

So far we have used a TCP port socket, because it's simpler, but in fact it's better to use Unix sockets than ports - there's less overhead.

Edit `mysite_nginx.conf`, changing it to match:

```
server unix:///path/to/your/mysite/mysite.sock; # for a file socket
# server 127.0.0.1:8001; # for a web port socket (we'll use this first)
```

and restart nginx.

Run uWSGI again:

```
uwsgi --socket mysite.sock --wsgi-file test.py
```

This time the `socket` option tells uWSGI which file to use.

Try http://example.com:8000/ in the browser.

If that doesn't work

Check your nginx error log(/var/log/nginx/error.log). If you see something like:

```
connect() to unix:///path/to/your/mysite/mysite.sock failed (13: Permission
denied)
```

then probably you need to manage the permissions on the socket so that nginx is allowed to use it.

Try:

```
uwsgi --socket mysite.sock --wsgi-file test.py --chmod-socket=666 # (very permissive)
```

or:

```
uwsgi --socket mysite.sock --wsgi-file test.py --chmod-socket=664 # (more sensible)
```

You may also have to add your user to nginx's group (which is probably www-data), or vice-versa, so that nginx can read and write to your socket properly.

It's worth keeping the output of the nginx log running in a terminal window so you can easily refer to it while troubleshooting.

4.2.8 Running the Django application with uswgi and nginx

Let's run our Django application:

```
uwsgi --socket mysite.sock --module mysite.wsgi --chmod-socket=664
```

Now uWSGI and nginx should be serving up not just a "Hello World" module, but your Django project.

4.2.9 Configuring uWSGI to run with a .ini file

We can put the same options that we used with uWSGI into a file, and then ask uWSGI to run with that file. It makes it easier to manage configurations.

Create a file called `mysite_uwsgi.ini`:

```
# mysite_uwsgi.ini file
[uwsgi]

# Django-related settings
# the base directory (full path)
chdir           = /path/to/your/project
# Django's wsgi file
module          = project.wsgi
# the virtualenv (full path)
home            = /path/to/virtualenv

# process-related settings
# master
master          = true
# maximum number of worker processes
processes       = 10
# the socket (use the full path to be safe
socket          = /path/to/your/project/mysite.sock
# ... with appropriate permissions - may be needed
# chmod-socket    = 664
# clear environment on exit
vacuum          = true
```

And run uswgi using this file:

```
uwsgi --ini mysite_uwsgi.ini # the --ini option is used to specify a file
```

Once again, test that the Django site works as expected.

4.2.10 Install uWSGI system-wide

So far, uWSGI is only installed in our virtualenv; we'll need it installed system-wide for deployment purposes.

Deactivate your virtualenv:

```
deactivate
```

and install uWSGI system-wide:

```
sudo pip install uwsgi

# Or install LTS (long term support).
pip install http://projects.unbit.it/downloads/uwsgi-lts.tar.gz
```

The uWSGI wiki describes several installation procedures. Before installing uWSGI system-wide, it's worth considering which version to choose and the most apppropriate way of installing it.

Check again that you can still run uWSGI just like you did before:

```
uwsgi --ini mysite_uwsgi.ini # the --ini option is used to specify a file
```

4.2.11 Emperor mode

uWSGI can run in 'emperor' mode. In this mode it keeps an eye on a directory of uWSGI config files, and will spawn instances ('vassals') for each one it finds.

Whenever a config file is amended, the emperor will automatically restart the vassal.

```
# create a directory for the vassals
sudo mkdir /etc/uwsgi
sudo mkdir /etc/uwsgi/vassals
# symlink from the default config directory to your config file
sudo ln -s /path/to/your/mysite/mysite_uwsgi.ini /etc/uwsgi/vassals/
# run the emperor
uwsgi --emperor /etc/uwsgi/vassals --uid www-data --gid www-data
```

You may need to run uWSGI with sudo:

```
sudo uwsgi --emperor /etc/uwsgi/vassals --uid www-data --gid www-data
```

The options mean:

- `emperor`: where to look for vassals (config files)
- `uid`: the user id of the process once it's started
- `gid`: the group id of the process once it's started

Check the site; it should be running.

4.2.12 Make uWSGI startup when the system boots

The last step is to make it all happen automatically at system startup time.

Edit `/etc/rc.local` and add:

```
/usr/local/bin/uwsgi --emperor /etc/uwsgi/vassals --uid www-data --gid www-data
```

before the line "exit 0".

And that should be it!

4.2.13 Further configuration

It is important to understand that this has been a *tutorial*, to get you started. You **do** need to read the nginx and uWSGI documentation, and study the options available before deployment in a production environment.

Both nginx and uWSGI benefit from friendly communities, who are able to offer invaluable advice about configuration and usage.

nginx

General configuration of nginx is not within the scope of this tutorial though you'll probably want it to listen on port 80, not 8000, for a production website.

You also ought to consider at having a separate server for non-Django serving, of static files for example.

uWSGI

uWSGI supports multiple ways to configure it. See uWSGI's documentation and examples.

Some uWSGI options have been mentioned in this tutorial; others you ought to look at for a deployment in production include (listed here with example settings):

```
env = DJANGO_SETTINGS_MODULE=mysite.settings # set an environment variable
pidfile = /tmp/project-master.pid # create a pidfile
harakiri = 20 # respawn processes taking more than 20 seconds
limit-as = 128 # limit the project to 128 MB
max-requests = 5000 # respawn processes after serving 5000 requests
daemonize = /var/log/uwsgi/yourproject.log # background the process & log
```

4.3 Running uWSGI on Dreamhost shared hosting

Note: the following tutorial gives suggestions on how to name files with the objective of hosting multiple applications on your account. You are obviously free to change naming schemes.

The tutorial assumes a shared hosting account, but it works on the VPS offer too (even if on such a system you have lot more freedom and you could use better techniques to accomplish the result)

4.3.1 Preparing the environment

Log in via ssh to your account and move to the home (well, you should be already there after login).

Download a uWSGI tarball (anything >= 1.4 is good, but for maximum performance use >= 1.9), explode it and build it normally (run make).

At the end of the procedure copy the resulting `uwsgi` binary to your home (just to avoid writing longer paths later).

Now move to the document root of your domain (it should be named like the domain) and put a file named `uwsgi.fcgi` in it with that content:

```
#!/bin/sh
/home/XXX/uwsgi /home/XXX/YYY.ini
```

change XXX with your account name and YYY with your domain name (it is only a convention, if you know what you are doing feel free to change it)

Give the file 'execute' permission

```
chmod +x uwsgi.fcgi
```

Now in your home create a YYY.ini (remember to change YYY with your domain name) with that content

```
[uwsgi]
flock = /home/XXX/YYY.ini
account = XXX
domain = YYY

protocol = fastcgi
master = true
processes = 3
logto = /home/%(account)/%(domain).uwsgi.log
virtualenv = /home/%(account)/venv
module = werkzeug.testapp:test_app
touch-reload = %p
auto-procname = true
procname-prefix-spaced = [%(domain)]
```

change the first three lines accordingly.

4.3.2 Preparing the python virtualenv

As we want to run the werkzeug test app, we need to install its package in a virtualenv.

Move to the home:

```
virtualenv venv
venv/bin/easy_install werkzeug
```

4.3.3 The .htaccess

Move again to the document root to create the .htaccess file that will instruct Apache to forward request to uWSGI

```
RewriteEngine On
RewriteBase /
RewriteRule ^uwsgi.fcgi/ - [L]
RewriteRule ^(.*)$ uwsgi.fcgi/$1 [L]
```

4.3.4 Ready

Go to your domain and you should see the Werkzeug test page. If it does not show you can check uWSGI logs in the file you specified with the logto option.

4.3.5 The flock trick

As the apache mod_fcgi/mod_fastcgi/mod_fcgid implemenetations are very flaky on process management, you can easily end with lot of copies of the same process running. The flock trick avoid that. Just remember that the flock option is very special as you cannot use placeholder or other advanced techniques with it. You can only specify the absolute path of the file to lock.

4.3.6 Statistics

As always remember to use uWSGI internal stats system

first, install uwsgitop

```
venv/bin/easy_install uwsgitop
```

Enable the stats server on the uWSGI config

```
[uwsgi]
flock = /home/XXX/YYY.ini
account = XXX
domain = YYY

protocol = fastcgi
master = true
processes = 3
logto = /home/%(account)/%(domain).uwsgi.log
virtualenv = /home/%(account)/venv
module = werkzeug.testapp:test_app
touch-reload = %p
auto-procname = true
procname-prefix-spaced = [%(domain)]

stats = /home/%(account)/stats_%(domain).sock
```

(as we have touch-reload in place, as soon as you update the ini file your instance is reloaded, and you will be able to suddenly use uwsgitop)

```
venv/bin/uwsgitop /home/WWW/stats_YYY.sock
```

(remember to change XXX and YYY accordingly)

4.3.7 Running Perl/PSGI apps (requires uWSGI >= 1.9)

Older uWSGI versions does not work well with plugins other than the python one, as the fastcgi implementation has lot of limits.

Starting from 1.9, fastCGI is a first-class citizen in the uWSGI project, so all of the plugins work with it.

As before, compile the uWSGI sources but this time we will build a PSGI monolithic binary:

```
UWSGI_PROFILE=psgi make
```

copy the resulting binary in the home as uwsgi_perl

Now edit the previously created uwsgi.fcgi file changing it to

```
#!/bin/sh
/home/XXX/uwsgi_perl /home/XXX/YYY.ini
```

(again, change XXX and YYY accordingly)

Now upload an app.psgi file in the document root (this is your app)

```
my $app = sub {
   my $env = shift;
   return [
```

```
        '200',
        [ 'Content-Type' => 'text/plain' ],
        [ "Hello World" ]
    ];
};
```

and change the uWSGI ini file accordingly

```
[uwsgi]
flock = /home/XXX/YYY.ini
account = XXX
domain = YYY

psgi = /home/%(account)/%(domain)/app.psgi
fastcgi-modifier1 = 5

protocol = fastcgi
master = true
processes = 3
logto = /home/%(account)/%(domain).uwsgi.log
virtualenv = /home/%(account)/venv
touch-reload = %p
auto-procname = true
procname-prefix-spaced = [%(domain)]

stats = /home/%(account)/stats_%(domain).sock
```

The only difference from the python one, is the usage of 'psgi' instead of 'module' and the addition of fastcgi-modifier1 that set the uWSGI modifier to the perl/psgi one

4.3.8 Running Ruby/Rack apps (requires uWSGI >= 1.9)

By default you can use passenger on Dreamhost servers to host ruby/rack applications, but you may need a more advanced application servers for your work (or you may need simply more control over the deployment process)

As the PSGI one you need a uWSGI version >= 1.9 to get better (and faster) fastcgi support

Build a new uWSGI binary with rack support

```
UWSGI_PROFILE=rack make
```

and copy it in the home as ''uwsgi_ruby''

Edit (again) the uwsgi.fcgi file changing it to

```
#!/bin/sh
/home/XXX/uwsgi_rack /home/XXX/YYY.ini
```

and create a Rack application in the document root (call it app.ru)

```
class RackFoo

     def call(env)
             [200, { 'Content-Type' => 'text/plain'}, ['ciao']]
     end

end

run RackFoo.new
```

Finally change the uWSGI .ini file for a rack app:

```
[uwsgi]
flock = /home/XXX/YYY.ini
account = XXX
domain = YYY

rack = /home/%(account)/%(domain)/app.ru
fastcgi-modifier1 = 7

protocol = fastcgi
master = true
processes = 3
logto = /home/%(account)/%(domain).uwsgi.log
virtualenv = /home/%(account)/venv
touch-reload = %p
auto-procname = true
procname-prefix-spaced = [%(domain)]

stats = /home/%(account)/stats_%(domain).sock
```

Only differences from the PSGI one, is the use of 'rack' instead of 'psgi', and the modifier1 mapped to 7 (the ruby/rack one)

4.3.9 Serving static files

It is unlikely you will need to serve static files on uWSGI on a dreamhost account. You can directly use apache for that (eventually remember to change the .htaccess file accordingly)

4.4 Running python webapps on Heroku with uWSGI

Prerequisites: a Heroku account (on the cedar platform), git (on the local system) and the heroku toolbelt.

Note: you need a uWSGI version >= 1.4.6 to correctly run python apps. Older versions may work, but are not supported.

4.4.1 Preparing the environment

On your local system prepare a directory for your project:

```
mkdir uwsgi-heroku
cd uwsgi-heroku
git init .
heroku create
```

the last command will create a new heroku application (you can check it on the web dashboard).

For our example we will run the Werkzeug WSGI testapp, so we need to install the werkzeug package in addition to uWSGI.

First step is creating a requirements.txt file and tracking it with git.

The content of the file will be simply

```
uwsgi
werkzeug
```

Let's track it with git

```
git add requirements.txt
```

4.4.2 Creating the uWSGI config file

Now we can create our uWSGI configuration file. Basically all of the features can be used on heroku

```
[uwsgi]
http-socket = :$(PORT)
master = true
processes = 4
die-on-term = true
module = werkzeug.testapp:test_app
memory-report = true
```

as you can see this is a pretty standard configuration. The only heroku-required options are –http-socket and –die-on-term.

The first is required to bind the uWSGI socket to the port requested by the Heroku system (exported via the environment variable PORT we can access with $(PORT))

The second one (–die-on-term) is required to change the default behaviour of uWSGI when it receive a SIGTERM (brutal realod, while Heroku expect a shutdown)

The memory-report option (as we are in a memory contrained environment) is a good thing.

Remember to track the file

```
git add uwsgi.ini
```

4.4.3 Preparing for the first commit/push

We now need the last step: creating the Procfile.

The Procfile is a file describing which commands to start. Generally (with other deployment systems) you will use it for every additional process required by your app (like memcached, redis, celery...), but under uWSGI you can continue using its advanced facilities to manage them.

So, the Procfile, only need to start your uWSGI instance:

```
web: uwsgi uwsgi.ini
```

Track it

```
git add Procfile
```

And finally let's commit all:

```
git commit -a -m "first commit"
```

and push it (read: deploy) to Heroku:

```
git push heroku master
```

The first time it will requires a couple of minutes as it need to prepare your virtualenv and compile uWSGI.

Following push will be much faster.

4.4.4 Checking your app

Running `heroku logs` you will be able to access uWSGI logs. You should get all of your familiar information, and eventually some hint in case of problems.

4.4.5 Using another version of python

Heroku supports different python versions. By default (currently, february 2013), Python 2.7.3 is enabled.

If you need another version just create a runtime.txt in your repository with a string like that:

```
python-2.7.2
```

to use python 2.7.2

Remember to add/commit that in the repository.

Every time you change the python version, a new uWSGI binary is built.

4.4.6 Multiprocess or Multithread ?

It obviosuly depend on your app. But as we are on a memory-limited environment you can expect better memory usage with threads.

In addition to this, if you plan to put production-apps on Heroku be sure to understand how Dynos and their proxy works (it is very important. really)

4.4.7 Async/Greethreads/Coroutine ?

As always, do not trust people suggesting you to ALWAYS use some kind of async mode (like gevent). If your app is async-friendly you can obviously use gevent (it is built by default in recent uWSGI releases), but if you do not know that, remain with multiprocess (or multithread).

4.4.8 Harakiri

As said previously, if you plan to put production-apps on heroku, be sure to understand how dynos and their proxy works. Based on that, try to always set the harakiri parameters to a good value for your app. (do not ask for a default value, IT IS APP-DEPENDENT)

4.4.9 Static files

Generally, serving static files on Heroku is not a good idea (mainly from a design point of view). You could obviously have that need. In such a case remember to use uWSGI facilities for that, in particular offloading is the best way to leave your workers free while you serve big files (in addition to this remember that your static files must be tracked with git)

4.4.10 Adaptive process spawning

None of the supported algorithm are good for the Heroku approach and, very probably, it makes little sense to use a dynamic process number on such a platform.

4.4.11 Logging

If you plan to use heroku on production, remember to send your logs (via udp for example) on an external server (with persistent storage).

Check the uWSGI available loggers. Surely one will fit your need. (pay attention to security, as logs will fly in clear).

UPDATE: a udp logger with crypto features is on work.

4.4.12 Alarms

All of the alarms plugin should work without problems

4.4.13 The Spooler

As your app runs on a non-persistent filesystem, using the Spooler is a bad idea (you will easily lose tasks).

4.4.14 Mules

They can be used without problems

4.4.15 Signals (timers, filemonitors, crons...)

They all works, but do not rely on cron facilities, as heroku can kill/destroy/restarts your instances in every moment.

4.4.16 External daemons

The –attach-daemon option and its –smart variants work without problems. Just remember you are on a volatile filesystem and you are not free to bind on port/addresses as you may wish

4.4.17 Monitoring your app (advanced/hacky)

Albeit Heroku works really well with newrelic services, you always need to monitor the internals of your uWSGI instance.

Generally you enable the stats subsystem with a tool like uwsgitop as the client.

You can simply add uwsgitop to you requirements.txt

```
uwsgi
uwsgitop
werkzeug
```

and enable the stats server on a TCP port (unix sockets will not work as the instance running uwsgitop is not on the same server !!!):

```
[uwsgi]
http-socket = :$(PORT)
master = true
processes = 4
die-on-term = true
module = werkzeug.testapp:test_app
```

```
memory-report = true
stats = :22222
```

Now we have a problem: how to reach our instance ?

We need to know the LAN address of the machine where our instance is phisically running. To accomplish that, a raw trick is running ifconfig on uWSGI startup:

```
[uwsgi]
http-socket = :$(PORT)
master = true
processes = 4
die-on-term = true
module = werkzeug.testapp:test_app
memory-report = true
stats = :22222
exec-pre-app = /sbin/ifconfig eth0
```

Now thanks to the `heroku logs` command you can know where your stats server is

```
heroku run uwsgitop 10.x.x.x:22222
```

change x.x.x with the discovered address and remember that you could not be able to bind on port 22222, so change it accordingly.

Is it worthy to make such a mess to get monitoring ? If you are testing your app before going to production, it could be a good idea, but if you plan to buy more dynos, all became so complex that you'd better to use some heroku-blessed technique (if any)

4.5 Running Ruby/Rack webapps on Heroku with uWSGI

Prerequisites: a Heroku account (on the cedar platform), git (on the local system) and the heroku toolbelt (or the old/deprecated heroku gem)

Note: you need a uWSGI version >= 1.4.8 to correctly run ruby/rack apps. Older versions may work, but are not supported.

4.5.1 Preparing the environment (a Sinatra application)

On your local system prepare the structure for your sinatra application

```
mkdir uwsgi-heroku
cd uwsgi-heroku
git init .
heroku create --stack cedar
```

the last command will create a new heroku application (you can check it on the web dashboard).

Next step is creating our Gemfile (this file containes the gem required by the application)

```
source 'https://rubygems.org'

gem "uwsgi"
gem "sinatra"
```

we now need to run `bundle install` to create the Gemfile.lock file

let's track the two with git:

```
git add Gemfile
git add Gemfile.lock
```

Finally create a config.ru file containing the Sinatra sample app

```
require 'sinatra'

get '/hi' do
  return "ciao"
end

run Sinatra::Application
```

and track it

```
git add config.ru
```

4.5.2 Creating the uWSGI config file

We are now ready to create the uWSGI configuration (we will use the .ini format in a file called uwsgi.ini).

The minimal setup for heroku is the following (check the comments in the file for an explanation)

```
[uwsgi]
; bind to the heroku required port
http-socket = :$(PORT)
; force the usage of the ruby/rack plugin for every request (7 is the official numbero for ruby/rack)
http-socket-modifier1 = 7
; load the bundler subsystem
rbrequire = bundler/setup
; load the application
rack = config.ru
; when the app receives the TERM signal let's destroy it (instead of brutal reloading)
die-on-term = true
```

but a better setup will be

```
[uwsgi]
; bind to the heroku required port
http-socket = :$(PORT)
; force the usage of the ruby/rack plugin for every request (7 is the official numbero for ruby/rack)
http-socket-modifier1 = 7
; load the bundler subsystem
rbrequire = bundler/setup
; load the application
rack = config.ru
; when the app receives the TERM signal let's destroy it (instead of brutal reloading)
die-on-term = true
; enable the master process
master = true
; spawn 4 processes to increase concurrency
processes = 4
; report memory usage after each request
memory-report = true
; reload if the rss memory is higher than 100M
reload-on-rss = 100
```

Let's track it

```
git add uwsgi.ini
```

4.5.3 Deploying to heroku

We need to create the last file (required by Heroku). It is the Procfile, that instruct the Heroku system on which process to start for a web application.

We want to spawn uwsgi (installed as a gem via bundler) using the uwsgi.ini config file

```
web: bundle exec uwsgi uwsgi.ini
```

track it

```
git add Procfile
```

And let's commit all:

```
git commit -a -m "first attempt"
```

And push to heroku:

```
git push heroku master
```

If all goes well, you will see your page under your app url on the /hi path

Remember to run `heroku logs` to check if all is ok.

4.5.4 fork() for dummies

uWSGI allows you to choose how to abuse the fork() syscall in your app.

By default the approach is loading the application in the master process and then fork() to the workers that will inherit a copy of the master process.

This approach speedup startup and can potentially consume less memory. The truth is that often (for the way ruby garbage collection works) you will get few memory gain. The real advantage in in performance as the vast majority of time during application startup is spent in (slowly) searching for files. With the fork() early approach you can avoid repeating that slow procedure one time for worker.

Obviously the uWSGI mantra is "do whatever you need, if you can't, it is a uWSGI bug" so if your app is not fork()-friendly you can add the `lazy-apps = true` option that will load your app one time per-worker.

4.5.5 The ruby GC

By default uWSGI, calls the ruby Garbage collector after each request. This ensure an optimal use of memory (remember on Heroku, your memory is limited) you should not touch the default approach, but if you experience a drop in performance you may want to tune it using the `ruby-gc-freq = n` option where n is the number of requests after the GC is called.

4.5.6 Concurrency

Albeit uWSGI supports lot of different paradigms for concurrency, the multiprocess one is suggested for the vast majority of ruby/rack apps.

Basically all popular ruby-frameworks rely on that. Remember that your app is limited so spawn a number of processes that can fit in your Heroku dyno.

Starting from uWSGI 1.9.14, native ruby 1.9/2.x threads support has been added. Rails4 (only in production mode !!!) supports them:

```
[uwsgi]
...
; spawn 8 threads per-process
threads = 8
; maps them as ruby threads
rbthreads = true
; do not forget to set production mode for rails4 apps !!!
env = RAILS_ENV=production
...
```

4.5.7 Harakiri

If you plan to put production-apps on heroku, be sure to understand how dynos and their proxy works. Based on that, try to always set the harakiri parameters to a good value for your app. (do not ask for a default value, IT IS APP-DEPENDENT)

Harakiri, is the maximum time a single request can run, before being destroyed by the master

4.5.8 Static files

Generally, serving static files on Heroku is not a good idea (mainly from a design point of view). You could obviously have that need. In such a case remember to use uWSGI facilities for that, in particular offloading is the best way to leave your workers free while you serve big files (in addition to this remember that your static files must be tracked with git)

Try to avoid serving static files from your ruby/rack code. It will be extremely slow (compared to the uWSGI facilities) and can hold your worker busy for the whole transfer of the file

4.5.9 Adaptive process spawning

None of the supported algorithms are good for the Heroku approach and, very probably, it makes little sense to use a dynamic process number on such a platform.

4.5.10 Logging

If you plan to use heroku on production, remember to send your logs (via udp for example) on an external server (with persistent storage).

Check the uWSGI available loggers. Surely one will fit your need. (pay attention to security, as logs will fly in clear).

UPDATE: a udp logger with crypto features is on work.

4.5.11 Alarms

All of the alarms plugin should work without problems

4.5.12 The Spooler

As your app runs on a non-persistent filesystem, using the Spooler is a bad idea (you will easily lose tasks).

4.5.13 Mules

They can be used without problems

4.5.14 Signals (timers, filemonitors, crons...)

They all works, but do not rely on cron facilities, as heroku can kill/destroy/restarts your instances in every moment.

4.5.15 External daemons

The –attach-daemon option and its –smart variants work without problems. Just remember you are on a volatile filesystem and you are not free to bind on port/addresses as you may wish

4.6 Reliably use FUSE filesystems for uWSGI vassals (with Linux)

Requirements: uWSGI 1.9.18, Linux kernel with FUSE and namespaces support.

FUSE is a technology allowing the implementation of filesystems in user space (hence the name: **F**ilesystem in **Use**rspace). There are hundreds of high-quality FUSE filesystems, so having your application relying on them is a common situation.

FUSE filesystems are normal system processes, so as any process in the system, they can crash (or you may involuntarily kill them). In addition to this, if you host multiple applications, each one requiring a FUSE mount point, you may want to avoid polluting the main mount points namespace and, more important, avoid having unused mount points in your system (i.e. an instance is completely removed and you do not want its FUSE mount point to be still available in the system).

The purpose of this tutorial is to configure an Emperor and a series of vassals, each one mounting a FUSE filesystem.

4.6.1 A Zip filesystem

fuse-zip is a FUSE process exposing a zip file as a filesystem.

Our objective is to store whole app in a zip archive and instruct uWSGI to mount it as a filesystem (via FUSE) under `/app`.

The Emperor

```
[uwsgi]
emperor = /etc/uwsgi/vassals
emperor-use-clone = fs,pid
```

The trick here is to use Linux namespaces to create vassals in a new pid and filesystem namespace.

The first one (`fs`) allows mount point created by the vassal to be available only to the vassal (without messing with the main system), while the `pid` allows the uWSGI master to be the "init" process (pid 1) of the vassal. Being "pid

1" means that when you die all your children die too. In our scenario (where our vassal launches a FUSE process on startup) it means that when the vassal is destroyed, the FUSE process is destroyed too, as well as its mount point.

A Vassal

```
[uwsgi]
uid = user001
gid = user001

; mount FUSE filesystem under /app (but only if it is not a reload)
if-not-reload =
  exec-as-user = fuse-zip -r /var/www/app001.zip /app
endif =

http-socket = :9090
psgi = /app/myapp.pl
```

Here we use the `-r` option of the `fuse-zip` command for a read-only mount.

Monitoring mount points

The problem with the current setup is that if the `fuse-zip` process dies, the instance will no more be able to access `/app` until it is respawned.

uWSGI 1.9.18 added the `--mountpoint-check` option. It forces the master to constantly verify the specified filesystem. If it fails, the whole instance will be brutally destroyed. As we are under The Emperor, soon after the vassal is destroyed it will be restarted in a clean state (allowing the FUSE mount point to be started again).

```
[uwsgi]
uid = user001
gid = user001

; mount FUSE filesystem under /app (but only if it is not a reload)
if-not-reload =
  exec-as-user = fuse-zip -r /var/www/app001.zip /app
endif =

http-socket = :9090
psgi = /app/myapp.pl

mountpoint-check = /app
```

4.6.2 Going Heavy Metal: A CoW rootfs (unionfs-fuse)

unionfs-fuse is a user-space implementation of a union filesystem. A union filesystem is a stack of multiple filesystems, so directories with same name are merged into a single view.

Union filesystems are more than this and one of the most useful features is copy-on-write (COW or CoW). Enabling CoWs means you will have an immutable/read-only mount point base and all of the modifications to it will go to another mount point.

Our objective is to have a read-only rootfs shared by all of our customers, and a writable mount point (configured as CoW) for each customer, in which every modification will be stored.

The Emperor

Previous Emperor configuration can be used, but we need to prepare our filesystems.

The layout will be:

```
/ufs (where we initially mount our unionfs for each vassal)
/ns
  /ns/precise (the shared rootfs, based on Ubuntu Precise Pangolin)
  /ns/lucid (an alternative rootfs for old-fashioned customers, based on Ubuntu Lucid Lynx)
  /ns/saucy (another shared rootfs, based on Ubuntu Saucy Salamander)

  /ns/cow (the customers' writable areas)
    /ns/cow/user001
    /ns/cow/user002
    /ns/cow/userXXX
    ...
```

We create our rootfs:

```
debootstrap precise /ns/precise
debootstrap lucid /ns/lucid
debootstrap saucy /ns/saucy
```

And we create the `.old_root` directory in each one (it is required for `pivot_root`, see below):

```
mkdir /ns/precise/.old_root
mkdir /ns/lucid/.old_root
mkdir /ns/saucy/.old_root
```

Be sure to install the required libraries into each of them (especially the libraries required for your language).

The `uwsgi` binary must be executable in this rootfs, so you have to invest a bit of time in it (a good approach is having a language plugin compiled for each distribution and placed into a common directory, for example, each rootfs could have an `/opt/uwsgi/plugins/psgi_plugin.so` file and so on).

A Vassal

Here things get a bit more complicated. We need to launch the unionfs process (as root, as it must be our new rootfs) and then call `pivot_root` (a more advanced `chroot` available on Linux).

Hooks are the best way to run custom commands (or functions) at various uWSGI startup phases.

In our example we will run FUSE processes at the "pre-jail" phase, and deal with mount points at the "as-root" phase (that happens after `pivot_root`).

```
[uwsgi]
; choose the approach that suits you best (plugins loading)
; this will be used for the first run ...
plugins-dir = /ns/precise/opt/uwsgi/plugins
; and this after a reload (where our rootfs is already /ns/precise)
plugins-dir = /opt/uwsgi/plugins
plugin = psgi

; drop privileges
uid = user001
gid = user001

; chdir to / to avoid problems after pivot_root
hook-pre-jail = callret:chdir /
```

```
; run unionfs-fuse using chroot (it is required to avoid deadlocks) and cow (we mount it under /ufs)
hook-pre-jail = exec:unionfs-fuse -ocow,chroot=/ns,default_permissions,allow_other /precise=RO:/cow/%

; change the rootfs to the unionfs one
; the .old_root directory is where the old rootfs is still available
pivot_root = /ufs /ufs/.old_root

; now we are in the new rootfs and in 'as-root' phase
; remount the /proc filesystem
hook-as-root = mount:proc none /proc
; bind mount the original /dev in the new rootfs (simplifies things a lot)
hook-as-root = mount:none /.old_root/dev /dev bind
; recursively un-mount the old rootfs
hook-as-root = umount:/.old_root rec,detach

; common bind
http-socket = :9090

; load the app (fix it according to your requirements)
psgi = /var/www/myapp.pl

; constantly check for the rootfs (seems odd but is is very useful)
mountpoint-check = /
```

If your app will try to write to its filesystem, you will see that all of the created/updated files are available in its `/cow` directory.

4.6.3 Notes

Some FUSE filesystems do not commit writes until they are unmounted. In such a case unmounting on vassal shutdown is a good trick:

```
[uwsgi]
; vassal options ...
...
; umount on exit
exec-as-user-atexit = fusermount -u /app
```

4.7 Build a dynamic proxy using RPC and internal routing

Work in progress (requires uWSGI 1.9.14, we use PyPy as the engine)

4.7.1 step 1: build your mapping function

we use the hostname as the mapping (you can use whatever you need)

```
import uwsgi

def my_mapper(hostname):
    return "127.0.0.1:3031"

uwsgi.register_rpc('the_mapper', my_mapper)
```

save it as myfuncs.py

4.7.2 step 2: building a routing table

```
[uwsgi]
; enable the pypy engine
pypy-home = /opt/pypy
; execute the myfuncs.py file (the 'the_mapper' rpc function will be registered)
pypy-exec = myfuncs.py

; bind to a port
http-socket = :9090

; let's define our routing table

; at every request (route-run execute the action without making check, use it instead of --route .*)
; and place the result in the MYNODE variable
route-run = rpcvar:MYNODE the_mapper ${HTTP_HOST}
; print the MYNODE variable (just for fun)
route-run = log:${MYNODE}
; proxy the request to the choosen backend node
route-run = http:${MYNODE}

; enable offloading for automagic non-blocking behaviour
; a good value for offloading is the number of cpu cores
offload-threads = 2
```

4.8 Setting up Graphite on Ubuntu using the Metrics subsystem

This tutorial will guide you in installing a multi-app server, with each application sending metrics to a central graphite/carbon server.

Graphite is available here: http://graphite.wikidot.com/

The uWSGI Metrics subsystem is documented here *The Metrics subsystem*

The tutorial assumes an Ubuntu Saucy (13.10) release on amd64

While for Graphite we will use Ubuntu official packages, uWSGI core and plugins will be downloaded and installed from official sources

4.8.1 Installing Graphite and the others needed packages

```
sudo apt-get install python-dev ruby-dev bundler build-essential libpcre3-dev graphite-carbon graphit
```

python-dev and ruby-dev are required as we want to support both WSGI and Rack apps.

pcre development headers allow you to build uWSGI with internal routing support (something you always want)

4.8.2 Initializing Graphite

The first step will be enabling th Carbon server.

The Graphite project is composed by three subsystems: whisper, carbon and the web frontend

Whisper is a data storage format (similar to rrdtool)

Carbon is the server gathering metrics and storing them in whisper files (well it does more, but this is its main purpose)

The web frontend visualize the charts/graphs built from the data gathered by the carbon server.

To enable the carbon server edit `/etc/default/graphite-carbon` and set CARBON_CACHE_ENABLED to true

Before starting the carbon server we need to build its search index.

Just run:

```
sudo /usr/bin/graphite-build-search-index
```

Then start the carbon server (at the next reboot it will be automatically started)

```
sudo /etc/init.d/carbon-cache start
```

4.8.3 Building and Installing uWSGI

Download latest stable uWSGI tarball

```
wget http://projects.unbit.it/downloads/uwsgi-latest.tar.gz
```

explode it, and from the created directory run:

```
python uwsgiconfig.py --build core
```

this will build the uWSGI "core" binary.

We now want to build the python, rack and carbon plugins:

```
python uwsgiconfig.py --plugin plugins/python core
python uwsgiconfig.py --plugin plugins/rack core
python uwsgiconfig.py --plugin plugins/carbon core
```

now we have `uwsgi`, `python_plugin.so`, `rack_plugin.so` and `carbon_plugin.so`

let's copy it to system directories:

```
sudo mkdir /etc/uwsgi
sudo mkdir /usr/lib/uwsgi
sudo cp uwsgi /usr/bin/uwsgi
sudo cp python_plugin.so /usr/lib/uwsgi
sudo cp rack_plugin.so /usr/lib/uwsgi
sudo cp carbon_plugin.so /usr/lib/uwsgi
```

4.8.4 Setting up the uWSGI Emperor

Create an upstart config file for starting *The uWSGI Emperor – multi-app deployment*

```
# Emperor uWSGI script

description "uWSGI Emperor"
start on runlevel [2345]
stop on runlevel [06]

exec /usr/bin/uwsgi --emperor /etc/uwsgi
```

save it as `/etc/init/emperor.conf` and start the Emperor:

```
start emperor
```

From now on, to start uWSGI instances just drop their config files into /etc/uwsgi

4.8.5 Spawning the Graphite web interface

Before starting the graphite web interface (that is a Django app) we need to initialize its database.

Just run:

```
sudo graphite-manage syncdb
```

this is the standard django syncdb command for manage.py. Just answer the questions to create an admin user.

Now we are ready to create a uWSGI vassal:

```
[uwsgi]
plugins-dir = /usr/lib/uwsgi
plugins = python
uid = _graphite
gid = _graphite
wsgi-file = /usr/share/graphite-web/graphite.wsgi
http-socket = :8080
```

Save it as `/etc/uwsgi/graphite.ini`

the _graphite user (and group) is created by the graphite ubuntu package. Our uWSGI vassal will run under this privileges.

The web interface will be available on the port 8080 of your server natively speaking HTTP. If you prefer to proxy it, just change `http-socket` to `http` or place it behind a full webserver like nginx (this step is not covered in this tutorial)

4.8.6 Spawning vassals sending metrics to Graphite

We are now ready to send applications metrics to the carbon/graphite server.

For every vassal file in /etc/uwsgi just be sure to add the following options:

```
[uwsgi]
...
plugins = carbon
enable-metrics = true
carbon-use-metrics = true
carbon-id = %n
carbon = 127.0.0.1:2003
...
```

The `carbon-id` set a meaningful prefix to each metric (%n automatically translates to the name without extension of the vassal file).

The `carbon` option set the address of the carbon server to send metrics to (by default the carbon server binds on port 2003, but you can change it editing `/etc/carbon/carbon.conf` and restarting the carbon server)

4.8.7 Using Graphiti (Ruby/Sinatra based) as alternative frontend

Graphiti is an alternative dashboard/frontend from Graphite writte in Sinatra (a Ruby/Rack framework).

Graphiti requires redis, so be sure a redis server is running in your system.

Running:

```
sudo apt-get install redis-server
```

will be enough

First step is cloning the graphiti app (place it where you want/need):

```
git clone https://github.com/paperlesspost/graphiti.git
```

then run the bundler tool (if you are not confident with the ruby world it is a tool for managing dependencies)

```
bundle install
```

Note: if the eventmachine gem installation fails, add "gem 'eventmachine"' in the Gemfile as the first gem and run bundle update. This will ensure latest eventmachine version will be installed

---

After bundle has installed all of the gems, you have to copy the graphiti example configuration:

```
cp config/settings.yml.example config/settings.yml
```

edit it and set graphite_base_url to the url where the graphite web interface (the django one) is running.

Now we can deploy it on uWSGI

```
[uwsgi]
plugins-dir = /usr/lib/uwsgi
plugins = rack
chdir = <path_to_graphiti>
rack = config.ru
rbrequire = bundler/setup
http-socket = :9191
uid = _graphite
gid = _graphite
```

save it as `/etc/uwsgi/graphiti.ini` to let the Emperor deploy it

You can now connect to port 9191 to manage your gathered metrics.

As always you are free to place the instance under a proxy.

4.8.8 Notes

By default the carbon server listens on a public address. Unless you know what you are doing you should point it to a local one (like 127.0.0.1)

uWSGI exports a gazillion of metrics (and more are planned), do not be afraid to use them

There is no security between apps and the carbon server, any apps can write metrics to it. If you are hosting untrusted apps you'd better to use other approcahes (like giving a graphite instance to every user in the system)

The same is true for redis, if you run untrusted apps a shared redis instance is absolutely not a good choice from a secuity point of view

CHAPTER 5

Articles

5.1 Serializing accept(), AKA Thundering Herd, AKA the Zeeg Problem

One of the historical problems in the UNIX world is the "thundering herd".

What is it?

Take a process binding to a networking address (it could be `AF_INET`, `AF_UNIX` or whatever you want) and then forking itself:

```
int s = socket(...)
bind(s, ...)
listen(s, ...)
fork()
```

After having forked itself a bunch of times, each process will generally start blocking on `accept()`

```
for(;;) {
    int client = accept(...);
    if (client < 0) continue;
    ...
}
```

The funny problem is that on older/classic UNIX, `accept()` is woken up in each process blocked on it whenever a connection is attempted on the socket.

Only one of those processes will be able to truly accept the connection, the others will get a boring `EAGAIN`.

This results in a vast number of wasted cpu cycles (the kernel scheduler has to give control to all of the sleeping processes waiting on that socket).

This behaviour (for various reasons) is amplified when instead of processes you use threads (so, you have multiple threads blocked on `accept()`).

The de facto solution was placing a lock before the `accept()` call to serialize its usage:

```
for(;;) {
    lock();
    int client = accept(...);
    unlock();
    if (client < 0) continue;
    ...
}
```

For threads, dealing with locks is generally easier but for processes you have to fight with system-specific solutions or fall back to the venerable SysV ipc subsystem (more on this later).

In modern times, the vast majority of UNIX systems have evolved, and now the kernel ensures (more or less) only one process/thread is woken up on a connection event.

Ok, problem solved, what we are talking about?

5.1.1 select()/poll()/kqueue()/epoll()/...

In the pre-1.0 era, uWSGI was a lot simpler (and less interesting) than the current form. It did not have the signal framework and it was not able to listen to multiple addresses; for this reason its loop engine was only calling `accept()` in each process/thread, and thundering herd (thanks to modern kernels) was not a problem.

Evolution has a price, so after a while the standard loop engine of a uWSGI process/thread moved from:

```
for(;;) {
    int client = accept(s, ...);
    if (client < 0) continue;
    ...
}
```

to a more complex:

```
for(;;) {
    int interesting_fd = wait_for_fds();
    if (fd_need_accept(interesting_fd)) {
        int client = accept(interesting_fd, ...);
        if (client < 0) continue;
    }
    else if (fd_is_a_signal(interesting_fd)) {
        manage_uwsgi_signal(interesting_fd);
    }
    ...
}
```

The problem is now the `wait_for_fds()` example function: it will call something like `select()`, `poll()` or the more modern `epoll()` and `kqueue()`.

These kinds of system calls are "monitors" for file descriptors, and they are woken up in all of the processes/threads waiting for the same file descriptor.

Before you start blaming your kernel developers, this is the right approach, as the kernel cannot know if you are waiting for those file descriptors to call `accept()` or to make something funnier.

So, welcome again to the thundering herd.

5.1.2 Application Servers VS WebServers

The popular, battle tested, solid, multiprocess reference webserver is Apache HTTPD.

It survived decades of IT evolutions and it's still one of the most important technologies powering the whole Internet.

Born as multiprocess-only, Apache had to always deal with the thundering herd problem and they solved it using SysV ipc semaphores.

(Note: Apache is really smart about that, when it only needs to wait on a single file descriptor, it only calls `accept()` taking advantage of modern kernels anti-thundering herd policies)

(Update: Apache 2.x even allows you to choose which lock technique to use, included flock/fcntl for very ancient systems, but on the vast majority of the system, when in multiprocess mode it will use the sysv semaphores)

Even on modern Apache releases, stracing one of its process (bound to multiple interfaces) you will see something like that (it is a Linux system):

```
semop(...); // lock
epoll_wait(...);
accept(...);
semop(...); // unlock
... // manage the request
```

the SysV semaphore protect your epoll_wait from thundering herd.

So, another problem solved, the world is a such a beatiful place... but ....

SysV IPC is not good for application servers :(*

The definition of "application server" is pretty generic, in this case we refer to one or more process/processes generated by an unprivileged (non-root) user binding on one or more network address and running custom, highly non-deterministic code.

Even if you had a minimal/basic knowledge on how SysV IPC works, you will know each of its components is a limited resource in the system (and in modern BSDs these limits are set to ridiculously low values, PostgreSQL FreeBSD users know this problem very well).

Just run 'ipcs' in your terminal to get a list of the allocated objects in your kernel. Yes, in your kernel. SysV ipc objects are persistent resources, they need to be removed manually by the user. The same user that could allocate hundreds of those objects and fill your limited SysV IPC memory.

One of the most common problems in the Apache world caused by the SysV ipc usage is the leakage when you brutally kills Apache instances (yes, you should never do it, but you don't have a choice if you are so brave/fool to host unreliable PHP apps in your webserver process).

To better understand it, spawn Apache and `killall -9 apache2`. Respawn it and run 'ipcs' you will get a new semaphore object every time. Do you see the problem? (to Apache gurus: yes I know there are hacky tricks to avoid that, but this is the default behaviour)

Apache is generally a system service, managed by a conscious sysadmin, so except few cases you can continue trusting it for more decades, even if it decides to use more SysV ipc objects :)

Your application server, sadly, is managed by different kind of users, from the most skilled one to the one who should change job as soon as possible to the one with the site cracked by a moron wanting to take control of your server.

Application servers are not dangerous, users are. And application servers are run by users. The world is an ugly place.

5.1.3 How application server developers solved it

Fast answer: they generally do not solve/care it

Note: we are talking about multiprocessing, we have already seen multithreading is easy to solve.

Serving static files or proxying (the main activities of a webserver) is generally a fast, non-blocking (very deterministic under various points of view) activity. Instead, a web application is way slower and heavier, so, even on moderately loaded sites, the amount of sleeping processes is generally low.

On highly loaded sites you will pray for a free process, and in non-loaded sites the thundering herd problem is completely irrelevant (unless you are running your site on a 386).

Given the relatively low number of processes you generally allocate for an application server, we can say thundering herd is a no-problem.

Another approach is dynamic process spawning. If you ensure your application server has always the minimum required number of processes running you will highly reduce the thundering herd problem. (check the family of –cheaper uWSGI options)

5.1.4 No-problem ??? So, again, what we are talking about ?

We are talking about "common cases", and for common cases there are a plethora of valid choices (instead of uWSGI, obviously) and the vast majority of problems we are talking about are non-existent.

Since the beginning of the uWSGI project, being developed by a hosting company where "common cases" do not exist, we cared a lot about corner-case problems, bizarre setups and those problems the vast majority of users never need to care about.

In addition to this, uWSGI supports operational modes only common/available in general-purpose webservers like Apache (I have to say Apache is probably the only general purpose webserver as it allows basically anything in its process space in a relatively safe and solid way), so lot of new problems combined with user bad-behaviour arise.

One of the most challenging development phase of uWSGI was adding multithreading. Threads are powerful, but are really hard to manage in the right way.

Threads are way cheaper than processes, so you generally allocate dozens of them for your app (remember, not used memory is wasted memory).

Dozens (or hundreds) of threads waiting for the same set of file descriptors bring us back to a thundering herd problem (unless all of your threads are constantly used).

For such a reason when you enable multiple threads in uWSGI a pthread mutex is allocated, serializing epoll()/kqueue()/poll()/select()... usage in each thread.

Another problem solved (and strange for uWSGI, without the need of an option ;)

But...

5.1.5 The Zeeg problem: Multiple processes with multiple threads

On June 27, 2013, David Cramer wrote an interesting blog post (you may not agree with its conclusions, but it does not matter now, you can continue hating uWSGI safely or making funny jokes about its naming choices or the number of options).

http://justcramer.com/2013/06/27/serving-python-web-applications/

The problem David faced was such a strong thundering herd that its response time was damaged by it (non constant performance was the main result of its tests).

Why did it happen? Wasn't the mutex allocated by uWSGI solving it?

David is (was) running uWSGI with 10 process and each of them with 10 threads:

```
uwsgi --processes 10 --threads 10 ...
```

While the mutex protects each thread in a single process to call `accept()` on the same request, there is no such mechanism (or better, it is not enabled by default, see below) to protect multiple processes from doing it, so given the number of threads (100) available for managing requests, it is unlikely that a single process is completely blocked (read: with all of its 10 threads blocked in a request) so welcome back to the thundering herd.

5.1.6 How David solved it ?

uWSGI is a controversial piece of software, no shame in that. There are users fiercely hating it and others morbidly loving it, but all agree that docs could be way better ([OT] it is good when all the people agree on something, but pull requests on uwsgi-docs are embarrassingly low and all from the same people.... come on, help us !!!)

David used an empirical approach, spotted its problem and decided to solve it running independent uwsgi processes bound on different sockets and configured nginx to round robin between them.

It is a very elegant approach, but it has a problem: nginx cannot know if the process on which is sending the request has all of its thread busy. It is a working but suboptimal solution.

The best way would be having an inter-process locking (like Apache), serializing all of the `accept()` in both threads and processes

5.1.7 uWSGI docs sucks: –thunder-lock

Michael Hood (you will find his name in the comments of David's post, too) signalled the problem in the uWSGI mailing-list/issue tracker some time ago, he even came out with an initial patch that ended with the `--thunder-lock` option (this is why open-source is better ;)

`--thunder-lock` is available since uWSGI 1.4.6 but never got documentation (of any kind)

Only the people following the mailing-list (or facing the specific problem) know about it.

5.1.8 SysV IPC semaphores are bad how you solved it ?

Interprocess locking has been an issue since uWSGI 0.0.0.0.0.1, but we solved it in the first public release of the project (in 2009).

We basically checked each operating system capabilities and chose the best/fastest ipc locking they could offer, filling our code with dozens of #ifdef.

When you start uWSGI you should see in its logs which "lock engine" has been chosen.

There is support for a lot of them:

- pthread mutexes with _PROCESS_SHARED and _ROBUST attributes (modern Linux and Solaris)
- pthread mutexes with _PROCESS_SHARED (older Linux)
- OSX Spinlocks (MacOSX, Darwin)
- Posix semaphores (FreeBSD >= 9)
- Windows mutexes (Windows/Cygwin)
- SysV IPC semaphores (fallback for all the other systems)

Their usage is required for uWSGI-specific features like caching, rpc and all of those features requiring changing shared memory structures (allocated with mmap() + _SHARED)

Each of these engines is different from the others and dealing with them has been a pain and (more important) some of them are not "ROBUST".

The "ROBUST" term is pthread-borrowed. If a lock is "robust", it means if the process locking it dies, the lock is released.

You would expect it from all of the lock engines, but sadly only few of them works reliably.

For this reason the uWSGI master process has to allocate an additional thread (the 'deadlock' detector) constantly checking for non-robust unreleased locks mapped to dead processes.

It is a pain, however, anyone will tell you IPC locking is easy should be accepted in a JEDI school...

5.1.9 uWSGI developers are fu*!ing cowards

Both David Cramer and Graham Dumpleton (yes, he is the mod_wsgi author but heavily contributed to uWSGI development as well to the other WSGI servers, this is another reason why open source is better) asked why `--thunder-lock` is not the default when multiprocess + multithread is requested.

This is a good question with a simple answer: we are cowards who only care about money.

uWSGI is completely open source, but its development is sponsored (in various way) by the companies using it and by Unbit.it customers.

Enabling "risky" features by default for a "common" usage (like multiprocess+multithread) is too much for us, and in addition to this, the situation (especially on linux) of library/kernel incompatibilities is a real pain.

As an example for having ROBUST pthread mutexes you need a modern kernel with a modern glibc, but commonly used distros (like the centos family) have a mix of older kernels with newer glibc and the opposite too. This leads to the inability to correctly detect which is the best locking engine for a platform, and so, when the uwsgiconfig.py script is in doubt it falls back to the safest approach (like non-robust pthread mutexes on linux).

The deadlock-detector should save you from most of the problem, but the "should" word is the key. Making a test suite (or even a single unit test) on this kind of code is basically impossible (well, at least for me), so we cannot be sure all is in the right place (and reporting threading bugs is hard for users as well as skilled developer, unless you work on pypy ;)

Linux pthread robust mutexes are solid, we are "pretty" sure about that, so you should be able to enable `--thunder-lock` on modern Linux systems with a 99.999999% success rates, but we prefer (for now) users consciously enable it

5.1.10 When SysV IPC semaphores are a better choice

Yes, there are cases on which SysV IPC semaphores gives you better results than system-specific features.

Marcin Deranek of Booking.com has been battle-testing uWSGI for months and helped us with fixing corner-case situations even in the locking area.

He noted system-specific lock-engines tend to favour the kernel scheduler (when choosing which process wins the next lock after an unlock) instead of a round-robin distribution.

As for their specific need for an equal distribution of requests among processes is better (they use uWSGI with perl, so no threading is in place, but they spawn lot of processes) they (currently) choose to use the "ipcsem" lock engine with:

```
uwsgi --lock-engine ipcsem --thunder-lock --processes 100 --psgi ....
```

The funny thing (this time) is that you can easily test if the lock is working well. Just start blasting the server and you will see in the request logs how the reported pid is different each time, while with system-specific locking the pids are pretty random with a pretty heavy tendency of favouring the last used process.

Funny enough, the first problem they faced was the ipcsem leakage (when you are in emergency, graceful reload/stop is your enemy and kill -9 will be your silver bullet)

To fix it, the –ftok option is available allowing you to give a unique id to the semaphore object and to reuse it if it is available from a previous run:

```
uwsgi --lock-engine ipcsem --thunder-lock --processes 100 --ftok /tmp/foobar --psgi ....
```

–ftok takes a file as an argument, it will use it to build the unique id. A common pattern is using the pidfile for it

5.1.11 What about other portable lock engines ?

In addition to "ipcsem", uWSGI (where available) adds "posixsem" too.

They are used by default only on FreeBSD >= 9, but are available on Linux too.

They are not "ROBUST", but they do not need shared kernel resources, so if you trust our deadlock detector they are a pretty-good approach. (Note: Graham Dumpleton pointed me to the fact they can be enabled on Apache 2.x too)

5.1.12 Conclusions

You can have the best (or the worst) software of the whole universe, but without docs it does not exist.

The Apache team still slam the face of the vast majority of us trying to touch their market share :)

5.1.13 Bonus chapter: using the Zeeg approach in a uWSGI friendly way

I have to admit, I am not a big fan of supervisord. It is a good software without doubts, but I consider the Emperor and the –attach-daemon facilities a better approach to the deployment problems. In addition to this, if you want to have a "scriptable"/"extendable" process supervisor I think Circus (http://circus.readthedocs.org/) is a lot more fun and capable (the first thing I have done after implementing socket activation in the uWSGI Emperor was making a pull request [merged, if you care] for the same feature in Circus).

Obviously supervisord works and is used by lot of people, but as a heavy uWSGI user I tend to abuse its features to accomplish a result.

The first approach I would use is binding to 10 different ports and mapping each of them to a specific process:

```
[uwsgi]
processes = 5
threads = 5

; create 5 sockets
socket = :9091
socket = :9092
socket = :9093
socket = :9094
socket = :9095

; map each socket (zero-indexed) to the specific worker
map-socket = 0:1
map-socket = 1:2
map-socket = 2:3
map-socket = 3:4
map-socket = 4:5
```

Now you have a master monitoring 5 processes, each one bound to a different address (no `--thunder-lock` needed)

For the Emperor fanboys you can make such a template (call it foo.template):

```
[uwsgi]
processes = 1
threads = 10
socket = :%n
```

Now make a symbolic link for each instance+port you want to spawn:

```
ln -s foo.template 9091.ini
ln -s foo.template 9092.ini
ln -s foo.template 9093.ini
ln -s foo.template 9094.ini
ln -s foo.template 9095.ini
ln -s foo.template 9096.ini
```

5.1.14 Bonus chapter 2: securing SysV IPC semaphores

My company hosting platform in heavily based on Linux cgroups and namespaces.

The first (cgroups) are used to limit/account resource usage, while the second (namespaces) are used to give an "isolated" system view to users (like seeing a dedicated hostname or root filesystem).

As we allow users to spawn PostgreSQL instances in their accounts we need to limit SysV objects.

Luckily, modern Linux kernels have a namespace for IPC, so calling unshare(CLONE_NEWIPC) will create a whole new set (detached from the others) of IPC objects.

Calling `--unshare ipc` in customer-dedicated Emperors is a common approach. When combined with memory cgroup you will end with a pretty secure setup.

5.1.15 Credits:

Author: Roberto De Ioris

Fixed by: Honza Pokorny

5.2 The Art of Graceful Reloading

Author: Roberto De Ioris

The following article is language-agnostic, and albeit uWSGI-specific, some of its initial considerations apply to other application servers and platforms too.

All of the described techniques assume a modern (>= 1.4) uWSGI release with the master process enabled.

5.2.1 What is a "graceful reload"?

During the life-cycle of your webapp you will reload it hundreds of time.

You need reloading for code updates, you need reloading for changes in the uWSGI configuration, you need reloading to reset the state of your app.

Basically, reloading is one of the most simple, frequent and **dangerous** operation you do every time.

So, why "graceful"?

Take a traditional (and highly suggested) architecture: a proxy/load balancer (like nginx) forwards requests to one or more uWSGI daemons listening on various addresses.

If you manage your reloads as "stop the instance, start the instance", the time slice between two phases will result in a brutal disservice for your customers.

The main trick for avoiding it is: not closing the file descriptors mapped to the uWSGI daemon addresses and abusing the Unix `fork()` behaviour (read: file descriptors are inherited by default) to `exec()` the `uwsgi` binary again.

The result is your proxy enqueuing requests to the socket until the latter will be able to `accept()` them again, with the user/customer only seeing a little slowdown in the first response (the time required for the app to be fully loaded again).

Another important step of graceful reload is to avoid destroying workers/threads that are still managing requests. Obviously requests could be stuck, so you should have a timeout for running workers (in uWSGI it is called the "worker's mercy" and it has a default value of 60 seconds).

These kind of tricks are pretty easy to accomplish and basically all of the modern servers/application servers do it (more or less).

But, as always, the world is an ugly place and lot of problems arise, and the "inherited sockets" approach is often not enough.

5.2.2 Things go wrong

We have seen that holding the uWSGI sockets alive allows the proxy webserver to enqueue requests without spitting out errors to the clients. This is true only if your app restarts fast, and, sadly, this may not always happen.

Frameworks like Ruby on Rails or Zope start up really slow by default, your app could start up slowly by itself, or your machine could be so overloaded that every process spawn (`fork()`) takes ages.

In addition to this, your site could be so famous that even if your app restarts in a couple of seconds, the queue of your sockets could be filled up forcing the proxy server to raise an error.

Do not forget, your workers/threads that are still running requests could block the reload (for various reasons) for more seconds than your proxy server could tolerate.

Finally, you could have made an application error in your just-committed code, so uWSGI will not start, or will start sending wrong things or errors...

Reloads (brutal or graceful) can easily fail.

5.2.3 The listen queue

Let's start with the dream of every webapp developer: *success*.

Your app is visited by thousands of clients and you obviously make money with it. Unfortunately, it is a very complex app and requires 10 seconds to warm up.

During graceful reloads, you expect new clients to wait 10 seconds (best case) to start seeing contents, but, unfortunately, you have hundreds of concurrent requests, so first 100 customers will wait during the server warm-up, while the others will get an error from the proxy.

This happens because the default size of uWSGI's listen queue is 100 slots. Before you ask, it is an average value choosen by the maximum value allowed by default by your kernel.

Each operating system has a default limit (Linux has 128, for example), so before increasing it you need to increase your kernel limit too.

So, once your kernel is ready, you can increase the listen queue to the maximum number of users you expect to enqueue during a reload.

To increase the listen queue you use the `--listen <n>` option where `<n>` is the maximum number of slots.

To raise kernel limits, you should check your OS docs. Some examples:

- sysctl `kern.ipc.somaxconn` on FreeBSD
- `/proc/sys/net/core/somaxconn` on Linux.

Note: This is only one of the reasons to tune the listen queue, but do not blindly set it to huge values as a way to increase availability.

5.2.4 Proxy timeouts

This is another thing you need to check if your reloads take a lot of time.

Generally, proxies allow you to set two timeouts:

connect Maximum amount of time the proxy will wait for a successful connection.

read Maximum amount of time the server will be able to wait for data before giving up.

When tuning the reloads, only the "connection" timeout matters. This timeout enters the game in the time slice between uWSGI's bind to an interface (or inheritance of it) and the call to `accept()`.

5.2.5 Waiting instead of errors is good, no errors and no waiting is even better

This is the focus of this article. We have seen how to increase the tolerance of your proxy during application server reloading. The customers will wait instead of getting scary errors, but we all want to make money, so why force them to wait?

We want zero-downtime and zero-wait.

5.2.6 Preforking VS lazy-apps VS lazy

This is one of the controversial choices of the uWSGI project.

By default uWSGI loads the whole application in the first process and after the app is loaded it does `fork()` itself multiple times. This is the common Unix pattern, it may highly reduce the memory usage of your app, allows lot of funny tricks and on some languages may bring you a lot of headaches.

Albeit its name, uWSGI was born as a Perl application server (it was not called uWSGI and it was not open source), and in the Perl world preforking is generally the blessed way.

This is not true for a lot of other languages, platforms and frameworks, so before starting dealing with uWSGI you should choose how to manage `fork()` in your stack.

Seeing it from the "graceful reloading" point of view, preforking extremely speeds up things: your app is loaded only one time, and spawning additional workers will be really fast. Avoiding disk access for each worker of your stack will increase startup times, expecially for frameworks or languages doing a lot of disk access to find modules.

Unfortunately, the preforking approach forces you to reload the whole stack whenever you make code changes instead of reloading only the workers.

In addition to this, your app could need preforking, or could completely crash due to it because of the way it has been developed.

lazy-apps mode instead loads your application one time per worker. It will require about O(n) time to load it (where n is the number of workers), will very probably consume more memory, but will run in a more consistent and clean environment.

Remember: lazy-apps is different from lazy, the first one only instructs uWSGI to load the application one time per worker, while the second is more invasive (and generally discouraged) as it changes a lot of internal defaults.

The following approaches will show you how to accomplish zero-downtime/wait reloads in both preforking and lazy modes.

Note: Each approach has pros and cons, choose carefully.

5.2.7 Standard (default/boring) graceful reload (aka `SIGHUP`)

To trigger it, you can:

- send `SIGHUP` to the master
- write `r` to *The Master FIFO*
- use `--touch-reload` option
- call `uwsgi.reload()` API.

In preforking and lazy-apps mode, it will:

1. Wait for running workers.
2. Close all of the file descriptors except the ones mapped to sockets.
3. Call `exec()` on itself.

In lazy mode, it will:

1. Wait for running workers.
2. Restart all of them (this means you cannot change uWSGI options during this kind of reload).

Warning: lazy is discouraged!

Pros:

- easy to manage
- no corner-case problems
- no inconsistent states
- basically full reset of the instance.

Cons:

- the ones we seen before
- listen queue filling up
- stuck workers
- potentially long waiting times.

5.2.8 Workers reloading in lazy-apps mode

Requires `--lazy-apps` option.

To trigger it:

- write `w` to *The Master FIFO*
- use `--touch-workers-reload` option.

It will wait for running workers and then restart each of them.

Pros:

- avoids restarting the whole instance.

Cons:

- no user-experience improvements over standard graceful reload, it is only a shortcut for situation when code updates do not imply instance reconfiguration.

5.2.9 Chain reloading (lazy apps)

Requires `--lazy-apps` option.

To trigger it:

- write `c` to *The Master FIFO*
- use `--touch-chain-reload` option.

This is the first approach that improves user experience. When triggered, it will restart one worker at time, and the following worker is not reloaded until the previous one is ready to accept new requests.

Pros:

- potentially highly reduces waiting time for clients
- reduces the load of the machine during reloads (no multiple processes loading the same code).

Cons:

- only useful for code updates
- you need a good amount of workers to get a better user experience.

5.2.10 Zerg mode

Requires a zerg server or a zerg pool.

To trigger it, run the instance in zerg mode.

This is the first approach that uses multiple instances of the same application to increase user experience.

Zerg mode works by making use of the venerable "fd passing over Unix sockets" technique.

Basically, an external process (the zerg server/pool) binds to the various sockets required by your app. Your uWSGI instance, instead of binding by itself, asks the zerg server/pool to pass it the file descriptor. This means multiple unrelated instances can ask for the same file descriptors and work together.

Zerg mode was born to improve auto-scalability, but soon became one of the most loved approaches for zero-downtime reloading.

Now, examples.

Spawn a zerg pool exposing `127.0.0.1:3031` to the Unix socket `/var/run/pool1`:

```
[uwsgi]
master = true
zerg-pool = /var/run/pool1:127.0.0.1:3031
```

Now spawn one or more instances attached to the zerg pool:

```
[uwsgi]
; this will give access to 127.0.0.1:3031 to the instance
zerg = /var/run/pool1
```

When you want to make update of code or options, just spawn a new instance attached to the zerg, and shut down the old one when the new one is ready to accept requests.

The so-called "zerg dance" is a trick for automation of this kind of reload. There are various ways to accomplish it, the objective is to automatically "pause" or "destroy" the old instance when the new one is fully ready and able to accept requests. More on this below.

Pros:

- potentially the silver bullet
- allows instances with different options to cooperate for the same app.

Cons:

- requires an additional process
- can be hard to master
- reload requires copy of the whole uWSGI stack.

5.2.11 The Zerg Dance: Pausing instances

We all make mistakes, sysadmins must improve their skill of fast disaster recovery. Focusing on avoiding them is a waste of time. Unfortunately, we are all humans.

Rolling back deployments could be your life-safer.

We have seen how zerg mode allows us to have multiple instances asking on the same socket. In the previous section we used it to spawn a new instance working together with the old one. Now, instead of shutting down the old instance, why not "pause" it? A paused instance is like the standby mode of your TV. It consumes very few resources, but you can bring it back very quickly.

"Zerg Dance" is the battle-name for the procedure of continuos swapping of instances during reloads. Every reload results in a "sleeping" instance and a running one. Following reloads destroy the old sleeping instance and transform the old running to the sleeping one and so on.

There are literally dozens of ways to accomplish the "Zerg Dance", the fact that you can easily use scripts in your reloading procedures makes this approach extremely powerful and customizable.

Here we will see the one that requires zero scripting, it could be the less versatile (and requires at least uWSGI 1.9.21), but should be a good starting point for the improvements.

The Master FIFO is the best way to manage instances instead of relying on Unix signals. Basically, you write single-char commands to govern the instance.

The funny thing about the Master FIFOs is that you can have many of them configured for your instance and swap one with another very easily.

An example will clarify things.

We spawn an instance with 3 Master FIFOs: new (the default one), running and sleeping:

```
[uwsgi]
; fifo '0'
master-fifo = /var/run/new.fifo
; fifo '1'
master-fifo = /var/run/running.fifo
; fifo '2'
master-fifo = /var/run/sleeping.fifo
; attach to zerg
zerg = /var/run/pool1
; other options ...
```

By default the "new" one will be active (read: will be able to process commands).

Now we want to spawn a new instance, that once is ready to accept requests will put the old one in sleeping mode. To do it, we will use uWSGI's advanced hooks. Hooks allow you to "make things" at various phases of uWSGI's life cycle. When the new instance is ready, we want to force the old instance to start working on the sleeping FIFO and be in "pause" mode:

```
[uwsgi]
; fifo '0'
master-fifo = /var/run/new.fifo
; fifo '1'
master-fifo = /var/run/running.fifo
; fifo '2'
master-fifo = /var/run/sleeping.fifo
; attach to zerg
zerg = /var/run/pool1

; hooks

; destroy the currently sleeping instance
if-exists = /var/run/sleeping.fifo
  hook-accepting1-once = writefifo:/var/run/sleeping.fifo Q
endif =
; force the currently running instance to became sleeping (slot 2) and place it in pause mode
if-exists = /var/run/running.fifo
  hook-accepting1-once = writefifo:/var/run/running.fifo 2p
endif =
; force this instance to became the running one (slot 1)
hook-accepting1-once = writefifo:/var/run/new.fifo 1
```

The `hook-accepting1-once` phase is run one time per instance soon after the first worker is ready to accept requests. The `writefifo` command allows writing to FIFOs without failing if the other peers are not connected (this is different from a simple `write` command that would fail or completely block when dealing with bad FIFOs).

Note: Both features have been added only in uWSGI 1.9.21, with older releases you can use the `--hook-post-app` option instead of `--hook-accepting1-once`, but you will lose the "once" feature, so it will work reliably only in preforking mode.

Instead of `writefifo` you can use the shell variant: `exec:echo <string> > <fifo>`.

Now start running instances with the same config files over and over again. If all goes well, you should always end with two instances, one sleeping and one running.

Finally, if you want to bring back a sleeping instance, just do:

```
# destroy the running instance
echo Q > /var/run/running.fifo

# unpause the sleeping instance and set it as the running one
echo p1 > /var/run/sleeping.fifo
```

Pros:

- truly zero-downtime reload.

Cons:

- requires high-level uWSGI and Unix skills.

5.2.12 SO_REUSEPORT (Linux >= 3.9 and BSDs)

On recent Linux kernels and modern BSDs you may try `--reuse-port` option. This option allows multiple unrelated instances to bind on the same network address. You may see it as a kernel-level zerg mode. Basically, all of the Zerg approaches can be followed.

Once you add `--reuse-port` to you instance, all of the sockets will have the `SO_REUSEPORT` flag set.

Pros:

- similar to zerg mode, could be even easier to manage.

Cons:

- requires kernel support
- could lead to inconsistent states
- you lose ability to use TCP addresses as a way to avoid incidental multiple instances running.

5.2.13 The Black Art (for rich and brave people): master forking

To trigger it, write `f` to *The Master FIFO*.

This is the most dangerous of the ways to reload, but once mastered, it could lead to pretty cool results.

The approach is: call `fork()` in the master, close all of the file descriptors except the socket-related ones, and `exec()` a new uWSGI instance.

You will end with two specular uWSGI instances working on the same set of sockets.

The scary thing about it is how easy (just write a single char to the master FIFO) is to trigger it...

With a bit of mastery you can implement the zerg dance on top of it.

Pros:

- does not require kernel support nor an additional process
- pretty fast.

Cons:

- a whole copy for each reload
- inconstent states all over the place (pidfiles, logging, etc.: the master FIFO commands could help fix them).

5.2.14 Subscription system

This is probably the best approach when you can count on multiple servers. You add the "fastrouter" between your proxy server (e.g., nginx) and your instances.

Instances will "subscribe" to the fastrouter that will pass requests from proxy server (nginx) to them while load balancing and constantly monitoring all of them.

Subscriptions are simple UDP packets that instruct the fastrouter which domain maps to which instance or instances.

As you can subscribe, you can unsubscribe too, and this is where the magic happens:

```
[uwsgi]
subscribe-to = 192.168.0.1:4040:unbit.it
unsubscribe-on-graceful-reload = true
; all of the required options ...
```

Adding `unsubscribe-on-graceful-reload` will force the instance to send an "unsubscribe" packet to the fastrouter, so until it will not be back no request will be sent to it.

Pros:

- low-cost zero-downtime
- a KISS approach (*finally*).

Cons:

- requires a subscription server (like the fastrouter) that introduces overhead (even if we are talking about microseconds).

5.2.15 Inconsistent states

Sadly, most of the approaches involving copies of the whole instance (like Zerg Dance or master forking) lead to inconsistent states.

Take, for example, an instance writing pidfiles: when starting a copy of it, that pidfile will be overwritten.

If you carefully plan your configurations, you can avoid inconsistent states, but thanks to *The Master FIFO* you can manage some of them (read: the most common ones):

- `l` command will reopen logfiles
- `P` command will update all of the instance pidfiles.

5.2.16 Fighting inconsistent states with the Emperor

If you manage your instances with the *Emperor*, you can use its features to avoid (or reduce number of) inconsistent states.

Giving each instance a different symbolic link name will allow you to map files (like pidfiles or logs) to different paths:

```
[uwsgi]
logto = /var/log/%n.log
pidfile = /var/run/%n.pid
; and so on ...
```

5.2.17 Dealing with ultra-lazy apps (like Django)

Some applications or frameworks (like Django) may load the vast majority of their code only at the first request. This means that customer will continue to experience slowdowns during reload even when using things like zerg mode or similar.

This problem is hard to solve (impossible?) in the application server itself, so you should find a way to force your app to load itself ASAP. A good trick (read: works with Django) is to call the entry-point function (like the WSGI callable) in the app itself:

```
def application(environ, sr):
    sr('200 OK', [('Content-Type', 'text/plain')])
    yield "Hello"

application({}, lambda x, y: None)  # call the entry-point function
```

You may need to pass CGI vars to the environ to make a true request: it depends on the WSGI app.

5.2.18 Finally: Do not blindly copy & paste!

Please, turn on your brain and try to adapt shown configs to your needs, or invent new ones.

Each app and system is different from the others.

Experiment before making a choice.

5.2.19 References

The Master FIFO

Hooks

Zerg mode

The uWSGI FastRouter

uWSGI Subscription Server

5.3 Fun with Perl, Eyetoy and RaspberryPi

Author: Roberto De Ioris

Date: 20131207

5.3.1 Intro

This article is the result of various experiments aimed at improving uWSGI performance and usability on various area before the 2.0 release.

To follow the article you need:

- a raspberrypi (any model) with a Linux distro installed (i have used the standard raspbian)
- an Eyetoy webcam (the PS3 one)
- a websockets enabled browser (basically any serious browser)
- a bit of perl knowledge (really a bit, the perl code is less than 10 lines ;)
- patience (building uWSGI+psgi+coroae on the rpi requires 13 minutes)

5.3.2 uWSGI subsystems and plugins

The project makes use of the following uWSGI subsystems and plugins:

- *WebSocket supports*
- *SharedArea – share memory pages between uWSGI components* (for storing frames)
- *uWSGI Mules* (for gathering frames)
- *The Symcall plugin*
- *uWSGI Perl support (PSGI)*
- *uWSGI asynchronous/non-blocking modes (updated to uWSGI 1.9)* (optional, we use Coro::Anyevent but you can rely on standard processes, albeit you will need way more memory)

5.3.3 What we want to accomplish

We want our rpi to gather frames from the eyetoy and stream them to the various connected clients using websockets (and a canvas to show them)

The whole system must use few memory, few cpu cycles and should support a big number of clients (well, even 10 clients will be a success for the raspberrypi hardware ;)

5.3.4 Technical background

The eyetoy captures frames in YUYV format (known as YUV 4:2:2). It means we need 4 bytes for 2 pixels.

By default the resolution is set to 640x480, so each frame will need 614400 bytes

Once we have a frame we need to decode it as RGBA to allow the HTML5 canvas to show it.

The translation between YUYV and RGBA is pretty heavy for the rpi (expecially if you need to do it for every connected client) so we will do it in the browser using javascript (well there are other approaches we can follow, just check the end of the article for them)

The uWSGI stack is composed by a mule gathering frames from the eyetoy and writing them to a sharedarea.

Workers constantly read from that sharedarea and send frames as websockets binary messages.

5.3.5 Let's start: the uwsgi-capture plugin

uWSGI 1.9.21 introduced a simplified (and safe) procedure to build uWSGI plugins. (so expect more third party plugins soon).

The project at: https://github.com/unbit/uwsgi-capture

shows a very simple plugin using the video4linux 2 api to gather frames.

Each frame is written in a sharedarea initialized by the plugin itself.

The first step is getting uWSGI and building it with the 'coroae' profile:

```
sudo apt-get install git build-essential libperl-dev libcoro-perl
git clone https://github.com/unbit/uwsgi
cd uwsgi
make coroae
```

the whole procedure requires 13 minutes, if all goes well you can clone the uwsgi-capture plugin and build it

```
git clone https://github.com/unbit/uwsgi-capture
./uwsgi --build-plugin uwsgi-capture
```

you now have the capture_plugin.so file in your uwsgi directory.

Plug your eyetoy to a usb port on your rpi and check if it works:

```
./uwsgi --plugin capture --v4l-capture /dev/video0
```

(the –v4l-capture option is exposed by the capture plugin)

If all goes well you should see the following lines in uWSGI startup logs:

```
/dev/video0 detected width = 640
/dev/video0 detected height = 480
/dev/video0 detected format = YUYV
```

```
sharedarea 0 created at 0xb6935000 (150 pages, area at 0xb6936000)
/dev/video0 started streaming frames to sharedarea 0
```

(the sharedarea memory pointers could be obviously different)

the uWSGI process will exit soon after them as we did not tell it what to do :)

The uwsgi-capture plugin exposes 2 functions:

captureinit() -> mapped as the init() hook of the plugin, it will be called automatically by uWSGI. If –v4l-capture option is specified, this function will initialize the specified device and will map it to a uWSGI sharedarea.

captureloop() -> this is the function gathering frames and writing them to the sharedarea. This function should constantly run (even if there are no clients reading frames)

We want a mule to run the captureloop() function:

```
./uwsgi --plugin capture --v4l-capture /dev/video0 --mule="captureloop()" --http-socket :9090
```

this time we have bound uWSGI to http port 9090 with a mule mapped to the “captureloop()” function. This mule syntax is exposed by the symcall plugin that take control of every mule argument ending with “()” (the quoting is required to avoid the shell making mess with parenthesis)

If all goes well you should see your uWSGI server spawning a master, a mule and a worker.

5.3.6 Step 2: the PSGI app

Time to write our websockets server sending eyetoy frames (you can find sources for the example here: https://github.com/unbit/uwsgi-capture/tree/master/rpi-examples).

The PSGI app will be very simple:

```
use IO::File;
use File::Basename;

my $app = sub {
     my $env = shift;

     # websockets connection happens on /eyetoy
     if ($env->{PATH_INFO} eq '/eyetoy') {
             # complete the handshake
             uwsgi::websocket_handshake($env->{HTTP_SEC_WEBSOCKET_KEY}, $env->{HTTP_ORIGIN});
             while(1) {
                     # wait for updates in the sharedarea
                     uwsgi::sharedarea_wait(0, 50);
                     # send a binary websocket message directly from the sharedarea
                     uwsgi::websocket_send_binary_from_sharedarea(0, 0)
             }
     }
     # other requests generate the html
     else {
             return [200, ['Content-Type' => 'text/html'], new IO::File(dirname(__FILE__).'/eyetoy.ht
     }
}
```

The only interesting parts are:

```
uwsgi::sharedarea_wait(0, 50);
```

this function suspend the current request until the specified sharedarea (the 'zero' one) gets an update. As this function is basically a poller, the second argument specifies the polling frequency (in milliseconds). 50 milliseconds gave us good results (feel free to try with other values).

```
uwsgi::websocket_send_binary_from_sharedarea(0, 0)
```

this is a special function sending a websocket binary message directly from the sharedarea (yes, zero-copy). The first argument is the sharedarea id (the 'zero' one) and the second is the position in the sharedarea to start reading from (zero again, as we want a full frame)

5.3.7 Step 3: HTML5

The html part (well it would be better to say the 'javascript' part) is very easy:

```
<html>
	<body>
		<canvas id="mystream" width="640" height="480" style="border:solid 1px red"></canvas>

		<script>


			var canvas = document.getElementById('mystream');
			var width = canvas.width;
			var height = canvas.height;
			var ctx = canvas.getContext("2d");
			var rgba = ctx.getImageData(0, 0, width, height);

			// fill alpha (optimization)
			for(y = 0; y< height; y++) {
				for(x = 0; x < width; x++) {
					pos = (y * width * 4) + (x * 4) ;
					rgba.data[pos+3] = 255;
				}
			}

			// connect to the PSGI websocket server
			var ws = new WebSocket('ws://' + window.location.host + '/eyetoy');
			ws.binaryType = 'arraybuffer';
			ws.onopen = function(e) {
				console.log('ready');
			};

			ws.onmessage = function(e) {
				var x, y;
				var ycbcr = new Uint8ClampedArray(e.data);
				// convert YUYV to RGBA
				for(y = 0; y< height; y++) {
					for(x = 0; x < width; x++) {
						pos = (y * width * 4) + (x * 4) ;
						var vy, cb, cr;
						if (x % 2 == 0) {
							ycbcr_pos = (y * width * 2) + (x * 2);
							vy = ycbcr[ycbcr_pos];
							cb = ycbcr[ycbcr_pos+1];
							cr = ycbcr[ycbcr_pos+3];
						}
						else {
```

```
                                        ycbcr_pos = (y * width * 2) + ((x-1) * 2);
                                        vy = ycbcr[ycbcr_pos+2];
                                        cb = ycbcr[ycbcr_pos+1];
                                        cr = ycbcr[ycbcr_pos+3];
                                    }
                                    var r = (cr + ((cr * 103) >> 8)) - 179;
                                    var g = ((cb * 88) >> 8) - 44 + ((cr * 183) >> 8) - 91;
                                    var b = (cb + ((cb * 198) >> 8)) - 227;
                                    rgba.data[pos] = vy + r;
                                    rgba.data[pos+1] = vy + g;
                                    rgba.data[pos+2] = vy + b;
                                }
                            }
                            // draw pixels
                            ctx.putImageData(rgba, 0, 0);
                        };
                        ws.onclose = function(e) { alert('goodbye');}
                        ws.onerror = function(e) { alert('oops');}
                </script>

    </body>
</html>
```

Nothing special here, the vast majority of the code is related to YUYV->RGBA conversion. Pay attention to set the websocket communication in 'binary' mode (binaryType = 'arraybuffer' is enough) and be sure to use a Uint8ClampedArray (otherwise performance will be terribly bad)

5.3.8 Ready to watch

```
./uwsgi --plugin capture --v4l-capture /dev/video0 --http-socket :9090 --psgi uwsgi-capture/rpi-examp
```

connect with your browser to tcp port 9090 of your raspberrypi and start watching

5.3.9 Concurrency

While you watch your websocket stream, you may want to start another browser window to see a second copy of your video. Unfortunately you spawned uWSGI with a single worker, so only a single client can get the stream.

You can add multiple workers easily:

```
./uwsgi --plugin capture --v4l-capture /dev/video0 --http-socket :9090 --psgi uwsgi-capture/rpi-examp
```

in this way up to 10 people will be able to watch the stream

But coroutines are way better (and cheaper) for such I/O bound applications:

```
./uwsgi --plugin capture --v4l-capture /dev/video0 --http-socket :9090 --psgi uwsgi-capture/rpi-examp
```

now we are able to manage 10 clients but with a single process !!! The rpi memory will be grateful to you.

5.3.10 Zero-copy all over the place

Why using the sharedarea ?

The sharedarea is one of the most advanced uWSGI features. If you give a look at the uwsgi-capture plugin you will see how it easily creates a sharedarea pointing to a mmap()'ed region. Basically each worker, thread (but please do not use threads with perl) or coroutine will have access to that memory in a concurrently-safe way.

In addition to this, thanks to the websocket-api -> sharedarea cooperation you can directly send websocket packets from a sharedarea without copying memory (except for the resulting websocket packet).

This is way faster than something like:

```
my $chunk = uwsgi::sharedarea_read(0, 0)
uwsgi::websocket_send_binary($chunk)
```

as we need to allocate the memory for $chunk at every iteration, copying the sharedarea content into it and finally encapsulating it in a websocket message.

With the sharedarea you remove the need to allocate (and free) memory constantly and to copy it from sharedarea to the perl vm.

5.3.11 Alternative approaches

There are obviously other approaches you can follow.

You could hack uwsgi-capture to allocate a second sharedarea in which it will directly write RGBA frames.

JPEG encoding is relatively fast, you can try encoding frames in the rpi and sending them as MJPEG frames (instead of using websockets):

```
my $writer = $responder->( [200, ['Content-Type' => 'multipart/x-mixed-replace; boundary=uwsgi_mjpeg_
$writer->write("--uwsgi_mjpeg_frame\r\n");
while(1) {
    uwsgi::sharedarea_wait(0);
    my $chunk = uwsgi::sharedarea_read(0, 0);
    $writer->write("Content-Type: image/jpeg\r\n");
    $writer->write("Content-Length: ".length($chunk)."\r\n\r\n");
    $writer->write($chunk);
    $writer->write("\r\n--uwsgi_mjpeg_frame\r\n");
}
```

5.3.12 Other languages

At the time of writing, the uWSGI PSGI plugin is the only one exposing the websockets+sharedarea additional api. The other languages plugins will be updated soon.

5.3.13 More hacking

The rpi board is really funny and uWSGI is a great companion for it (expecially its lower-level api functions).

As an exercise left to the reader: remember you can mmap() the address 0x0x20200000 to access the rpi GPIO controller...ready to write a uwsgi-gpio plugin ?

CHAPTER 6

uWSGI Subsystems

6.1 The uWSGI alarm subsystem (from 1.3)

As of 1.3, uWSGI includes an alarm system. This subsystem allows the developer/sysadmin to 'announce' special conditions of an app via various channels. For example, you may want to get notified via Jabber/XMPP of a full listen queue, or a harakiri condition. The alarm subsystem is based on two components: an event monitor and an event action.

An event monitor is something waiting for a specific condition (like an event on a file descriptor or a specific log message).

As soon as the condition is true an action (like sending an email) is triggered.

6.1.1 Embedded event monitors

Event monitors can be added via plugins, the uWSGI core includes the following:

`log-alarm` trigger alarm when a specific regexp matches a logline

`alarm-fd` trigger alarm when the specififed file descriptor is ready (low-level, it is the base of most of the alarm plugins)

`alarm-backlog` trigger alarm when the socket backlog queue is full

6.1.2 Defining an alarm

You can define an unlimited number of alarms. Each alarm has a unique name.

Currently the following alarm actions are available in the main distribution:

```
'cmd' run a command passing the log line to the stdin
'signal' generate a uwsgi signal
'mule' send the log line to a mule
'curl' pass the log line to a curl url (http,https and smtp are supported)
'xmpp' send the log line via XMPP/jabber
```

To define an alarm, use the option `--alarm`.

```
--alarm "<name> <plugin>:<opts>"
```

Remember to quote ONLY when you are defining alarms on the command line.

```
[uwsgi]
alarm = mailme cmd:mail -s 'uWSGI alarm' -a 'From: foobar@example.com' admin@example.com
alarm = cachefull signal:17
```

Here we define two alarms: `mailme` and `cachefull`. The first one invokes the `mail` binary to send the log line to a mail address; the second one generates an uWSGI signal. We now need to add rules to trigger alarms:

```
[uwsgi]
alarm = mailme cmd:mail -s 'uWSGI alarm' -a 'From: foobar@example.com' admin@example.com
alarm = cachefull signal:17
log-alarm = cachefull,mailme uWSGI listen queue of socket
log-alarm = mailme HARAKIRI ON WORKER
```

The syntax of log-alarm is

```
--log-alarm "<name> <regexp>"
```

In our previous example we defined two conditions using regexps applied to log lines. The first one will trigger both alarms when the listen queue is full, while the second will only invoke 'mailme' when a worker commits harakiri.

6.1.3 Damnit, this... this is the rawest thing I've seen...

You may be right. But if you throw away your "being a cool programmer with a lot of friends and zero money" book for a moment, you will realize just how many things you can do with such a simple system. Want an example?

```
[uwsgi]
alarm = jabber xmpp:foobar@jabber.xxx;mysecretpassword;admin@jabber.xxx,admin2@jabber.xxx
log-alarm = jabber ^TERRIBLE ALARM
```

Now in your app you only need to add

```
print "TERRIBLE ALARM the world exploded !!!"
```

to send a jabber message to `admin@jabber.xxx` and `admin2@jabber.xxx` without adding any significant overhead to your app (as alarms are triggered by one or more threads in the master process, without bothering workers).

How about another example?

Check this Rack middleware:

```
class UploadCheck
  def initialize(app)
    @app = app
  end

  def call(env)
    if env['REQUEST_METHOD'] == 'POST' and env['PATH_INFO'] == '/upload'
      puts "TERRIBLE ALARM an upload has been made"
    end
    @app.call(env)
  end
end
```

6.1.4 Protecting from bad rules

Such a versatile system could be open to a lot of ugly bugs, mainly infinite loops. Thus, try to build your regexps carefully. The embedded anti-loop subsystem should protect against loglines wrongly generated by alarm plugin. This system is not perfect so please double-check your regexps.

If you are building a plugin, be sure to prepend your log messages with the '[uwsgi-alarm' string. These lines will be skipped and directly passed to the log subsystem. A convenience API function is available: `uwsgi_log_alarm()`.

6.1.5 How does log-alarm work ?

Enabling log-alarm automatically puts the uWSGI instance in *log-master mode*, delegating log writes to the master. The alarm subsystem is executed by the master just before passing the log line to the log plugin. Blocking alarm plugins should run in a thread (like the curl and xmpp one), while the simple ones (like signal and cmd) may run directly in the master.

6.1.6 Available plugins and their syntax

cmd

Run a shell command, logline is passed to the stdin:

```
cmd:<command>
```

signal

Raise a [wiki:SignalFramework] uwsgi signal:

```
signal:[signum]
```

mule

Send the logline to a mule waiting for [wiki:Mules messages]

```
mule:[mule_id]
```

curl

Send logline to a curl url. This is not compiled in by default, so if you need to build it just run:

```
python uwsgiconfig.py --plugin plugins/alarm_curl
```

```
curl:<url>[;opt1=val1;opt2=val2]
```

`url` is a standard curl url, while the options currently exposed are

```
"url"
"mail_to"
"mail_from"
"subject"
"ssl"
"auth_user"
"auth_pass"
"method"
"timeout"
"conn_timeout"
```

So, for sending mail via SMTP AUTH:

```
[uwsgi]
plugins = alarm_curl
alarm = test curl:smtp://mail.example.com;mail_to=admin@example.com;mail_from=uwsgi@example.com;auth_
```

Or to POST the logline to an http server protected with basic auth:

```
[uwsgi]
plugins = alarm_curl
alarm = test2 curl:http://192.168.173.6:9191/argh;auth_user=topogigio;auth_pass=foobar
```

xmpp

Probably the most funny/interesting one. You need the `libgloox` package to build the xmpp alarm plugin (on Debian/Ubuntu, `apt-get install gloox-dev`).

```
python uwsgiconfig.py --plugin plugins/alarm_xmpp
```

```
xmpp:<jid>;<password>;<recipients>
```

You can set multiple recipients using ',' as delimiter.

```
[uwsgi]
plugins = alarm_xmpp
alarm = jabber xmpp:app@example.it;secret1;foo1@foo.it,foo2@foo.it
```

A funnier thing still about the XMPP plugin is that you will see the Jabber account of your app going down when your app dies...

Some XMPP servers (most notably the OSX server one) requires you to bind to a resource. You can do thus by appending /resource to the JID:

```
[uwsgi]
plugins = alarm_xmpp
alarm = jabber xmpp:max@server.local/uWSGI;secret1;foo1@foo.it,foo2@foo.it
```

speech

A toy plugin for OSX, used mainly for showing Objective-C integration with uWSGI. It simply uses the OSX speech synthesizer to 'announce' the alarm.

```
python uwsgiconfig.py --plugin plugins/alarm_speech
```

```
[uwsgi]
plugins = alarm_speech
http-socket = :8080
alarm = say speech:
log-alarm = say .*
```

Turn on your speakers, run uWSGI and start listening...

airbrake

Starting with 1.9.9 uWSGI includes the `--alarm-segfault` option to raise an alarm when uWSGI segfaults.

The `airbrake` plugin can be used to send segfault backtraces to airbrake compatible servers. Like Airbrake itself and its open source clone errbit (https://github.com/errbit/errbit), Airbrake support is experimental and it might not fully work in all cases.

```
plugins = airbrake
alarm = errbit airbrake:http://errbit.domain.com/notifier_api/v2/notices;apikey=APIKEY;subject=uWSGI
alarm-segfault = errbit
```

Note that alarm-segfault does not require airbrake plugin. A backtrace can be sent using any other alarm plugin.

6.2 The uWSGI caching framework

Note: This page is about "new-generation" cache introduced in uWSGI 1.9. For old-style cache (now simply named "web caching") check *WebCaching framework*

uWSGI includes a very fast, all-in-memory, zero-IPC, SMP-safe, constantly-optimizing, highly-tunable, key-value store simply called "the caching framework". A single uWSGI instance can create an unlimited number of "caches" each one with different setup and purpose.

6.2.1 Creating a "cache"

To create a cache you use the `--cache2` option. It takes a dictionary of arguments specifying the cache configuration. To have a valid cache you need to specify its name and the maximum number of items it can contains.

```
uwsgi --cache2 name=mycache,items=100 --socket :3031
```

this will create a cache named "mycache" with a maximum of 100 items. Each item can be at most 64k.

6.2.2 A sad/weird/strange/bad note about "the maximum number of items"

If you start with a 100-items cache you will suddenly note that the true maximum number of items you can use is indeed 99.

This is because the first item of the cache is always used as "NULL/None/undef" item internally.

Remember it, when you start planning your cache configuration

6.2.3 Configuring the cache (how it works)

The uWSGI cache works like a file system. You have an area for storing keys (metadata) followed by a series of fixed size blocks in which to store the content of each key. Another memory area, the hash table is allocated for fast search of keys. When you request a key, it is first hashed over the hash table. Each hash points to a key in the metadata area. Keys can be linked to manage hash collisions. Each key has a reference to the block containing its value.

6.2.4 Single block (faster) vs. bitmaps (slower)

Note: bitmap mode is considered production ready starting from uWSGI 2.0.2

In the standard ("single block") configuration a key can only map to a single block. Thus if you have a cache block size of 64k your items can be at most 65,535 bytes long. Conversely items smaller than that will still consume 64k of

memory. The advantage of this approach is its simplicity and speed. The system does not need to scan the memory for free blocks every time you insert an object in the cache.

If you need a more versatile (but relatively slower) approach, you can enable the "bitmap" mode. Another memory area will be created containing a map of all of the used and free blocks of the cache. When you insert an item the bitmap is scanned for contiguous free blocks. Blocks must be contiguous, this could lead to a bit of fragmentation but it is not as big a problem as with disk storage, and you can always tune the block size to reduce fragmentation.

6.2.5 Persistent storage

You can store cache data in a backing store file to implement persistence. As this is managed by `mmap()` it is almost transparent to the user. You should not rely on this for data safety (disk syncing is managed asynchronously); use it only for performance purposes.

6.2.6 Network access

All of your caches can be accessed over the network. A request plugin named "cache" (modifier1 111) manages requests from external nodes. On a standard monolithic build of uWSGI the cache plugin is always enabled. The cache plugin works in a fully non-blocking way, and it is greenthreads/coroutine friendly so you can use technologies like gevent or Coro::AnyEvent with it safely.

6.2.7 UDP sync

This technique has been inspired by the STUD project, which uses something like this for SSL session scaling (and coincidentally the same approach can be used with uWSGI SSL/HTTPS routers). Basically whenever you set/update/delete an item from the cache, the operation is propagated to remote nodes via simple UDP packets. There are no built-in guarantees with UDP syncing so use it only for very specific purposes, like *Scaling SSL connections (uWSGI 1.9)*.

6.2.8 –cache2 options

This is the list of all of the options (and their aliases) of `--cache2`.

name

Set the name of the cache. Must be unique in an instance.

max-items || maxitems || items

Set the maximum number of cache items.

blocksize

Set the size (in bytes) of a single block.

blocks

Set the number of blocks in the cache. Useful only in bitmap mode, otherwise the number of blocks is equal to the maximum number of items.

hash

Set the hash algorithm used in the hash table. Currentl options are “djb33x” (default) and “murmur2”.

hashsize || hash_size

this is the size of the hash table in bytes. Generally 65536 (the default) is a good value. Change it only if you know what you are doing or if you have a lot of collisions in your cache.

keysize || key_size

Set the maximum size of a key, in bytes (default 2048)

store

Set the filename for the persistent storage. If it doesn’t exist, the system assumes an empty cache and the file will be created.

store_sync || storesync

Set the number of seconds after which msync() is called to flush memory cache on disk when in persistent mode. By default it is disabled leaving the decision-making to the kernel.

node || nodes

A semicolon separated list of UDP servers which will receive UDP cache updates.

sync

A semicolon separated list of uwsgi addresses which the cache subsystem will connect to for getting a full dump of the cache. It can be used for initial cache synchronization. The first node sending a valid dump will stop the procedure.

udp || udp_servers || udp_server || udpserver

A semicolon separated list of UDP addresses on which to bind the cache to wait for UDP updates.

bitmap

Set to 1 to enable bitmap mode.

lastmod

Setting lastmod to 1 will update last_modified_at timestamp of each cache on every cache item modification. Enable it if you want to track this value or if other features depend on it. This value will then be accessible via the stats socket.

6.2.9 Accessing the cache from your applications using the cache api

You can access the various cache in your instance or on remote instances by using the cache API. Currently the following functions are exposed (each language might name them a bit differently from the standard):

- cache_get(key[,cache])
- cache_set(key,value[,expires,cache])
- cache_update(key,value[,expires,cache])
- cache_exists(key[,cache])
- cache_del(key[,cache])
- cache_clear([cache])

If the language/platform calling the cache API differentiates between strings and bytes (like Python 3 and Java) you have to assume that keys are strings and values are bytes (or bytearray in the java way). Otherwise keys and values are both strings in no specific encoding, as internally the cache values and keys are simple binary blobs.

The `expires` argument (default to 0 for disabled) is the number of seconds after the object is no more valid (and will be removed by the cache sweeper, see below)

The `cache` argument is the so called "magic identifier". Its syntax is `cache[@node]`.

To operate on the local cache "mycache" you set it as "mycache", while to operate on "yourcache" on the uWSGI server at 192.168.173.22 port 4040 the value will be `yourcache@192.168.173.22:4040`.

An empty cache value means the default cache which is generally the first initialized. The default value is empty.

All of the network operations are transparent, fully non-blocking, and threads/greenthreads friendly.

6.2.10 The Cache sweeper thread

When at least one cache is configured and the master is enabled a thread named "the cache sweeper" is started. Its main purpose is deleting expired keys from the cache. So, if you want auto-expiring you need to enable the master.

6.2.11 Web caching

In its first incarnation the uWSGI caching framework was meant only for caching of web pages. The old system has been rebuilt. It is now named *WebCaching framework*. Enabling the old-style `--cache` option will create a cache named "default".

6.3 WebCaching framework

Note: This is a port of the old caching subsystem to the new uWSGI caching API documented here *The uWSGI caching framework*. Using the options here will create a new-style cache named "default".

To enable web caching, allocate slots for your items using the `cache` option. The following command line would create a cache that can contain at most 1000 items.

```
./uwsgi --socket 127.0.0.1:3031 --module mysimpleapp --master --processes 4 --cache 1000
```

To use the cache in your application,

```
uwsgi.cache_set("foo_key", "foo_value") # set a key
value = uwsgi.cache_get("foo_key") # get a key.
```

6.3.1 Persistent storage

You can store cache data in a backing store file to implement persistence. Simply add the `cache-store <filename>` option. Every kernel will commit data to the disk at a different rate. You can set if/when to force this with `cache-store-sync <n>`, where `n` is the number of master cycles to wait before each disk sync.

6.3.2 Cache sweeper

Since uWSGI 1.2, cache item expiration is managed by a thread in the *master* process, to reduce the risk of deadlock. This thread can be disabled (making item expiry a no-op) with the `cache-no-expire` option.

The frequency of the cache sweeper thread can be set with `cache-expire-freq <seconds>`. You can make the sweeper log the number of freed items with `cache-report-freed-items`.

6.3.3 Directly accessing the cache from your web server

```
location / {
 uwsgi_pass 127.0.0.1:3031;
 uwsgi_modifier1 111;
 uwsgi_modifier2 3;
 uwsgi_param key $request_uri;
}
```

That's it! Nginx would now get HTTP responses from a remote uwsgi protocol compliant server. Although honestly this is not very useful, as if you get a cache miss, you will see a blank page.

A better system, that will fallback to a real uwsgi request would be

```
location / {
  uwsgi_pass 192.168.173.3:3032;
  uwsgi_modifier1 111;
  uwsgi_modifier2 3;
  uwsgi_param key $request_uri;
  uwsgi_pass_request_headers off;
  error_page 502 504 = @real;
}

location @real {
  uwsgi_pass 192.168.173.3:3032;
  uwsgi_modifier1 0;
  uwsgi_modifier2 0;
  include uwsgi_params;
}
```

6.3.4 Django cache backend

If you are running Django, there's a ready-to-use application called `django-uwsgi-cache`. It is maintained by Ionel Cristian Mărie at https://github.com/ionelmc/django-uwsgi-cache and availabe on pypi.

6.4 The uWSGI cron-like interface

uWSGI's *master* has an internal cron-like facility that can generate events at predefined times. You can use it

- via the uWSGI API, in which case cron events will generate uWSGI signals
- directly via options, in which case events will run shell commands

6.4.1 uWSGI signal based

The `uwsgi.add_cron()` function is the interface to the uWSGI signal cron facility. The syntax is

```
uwsgi.add_cron(signal, minute, hour, day, month, weekday)
```

The last 5 arguments work similarly to a standard crontab, but instead of "*", use -1, and instead of "*/2", "*/3", etc. use -2 and -3, etc.

```
import uwsgi

def five_o_clock_on_the_first_day_of_the_month(signum):
    print "It's 5 o'clock of the first day of the month."

uwsgi.register_signal(99, "", five_o_clock_on_the_first_day_of_the_month)
uwsgi.add_cron(99, 0, 5, 1, -1, -1)
```

6.4.2 Timers vs. cron

Recurring events not related to specific dates should use timers/rb_timers. When you are interested in a specific date/hour use cron.

For example:

```
uwsgi.add_cron(99, -1, -1, -1, -1, -1) # ugly, bad and inefficient way to run signal 99 every minute
uwsgi.add_timer(99, 60) # much better.
```

6.4.3 Notes

- `day` and `weekday` are ORed as the original crontab specifications.
- By default, you can define up to 64 signal-based cron jobs per master. This

value may be increased in `uwsgi.h`.

6.4.4 Option-based cron

You can define cron tasks directly in configuration with the `cron` option. You can specify an unlimited number of option-based cron records. The syntax is the same of the signal-based ones.

For example:

```
[uwsgi]
cron = 59 2 -1 -1 -1 /usr/bin/backup_my_home --recursive
cron = 9 11 -1 -1 2 /opt/dem/bin/send_reminders
```

```
<uwsgi>
    <cron>59 2 -1 -1 -1 /usr/bin/backup_my_home --recursive</cron>
    <cron>9 11 -1 -1 2 /opt/dem/bin/send_reminders</cron>
</uwsgi>
```

```
[uwsgi]
; every two hours
cron = -1 -2 -1 -1 -1 /usr/bin/backup_my_home --recursive
```

Legion crons

When your instance is part of a *The uWSGI Legion subsystem*, you can configure it to run crons only if it is the Lord of the specified Legion:

```
[uwsgi]
legion = mycluster 225.1.1.1:1717 100 bf-cbc:hello
legion-node = mycluster 225.1.1.1:1717
; every two hours
legion-cron = mycluster -1 -2 -1 -1 -1 /usr/bin/backup_my_home --recursive
```

Unique crons

Note: This feature is available since 1.9.11.

Some commands can take a long time to finish or just hang doing their thing. Sometimes this is okay, but there are also cases when running multiple instances of the same command can be dangerous.

For such cases the `unique-cron` and `unique-legion-cron` options were added. The syntax is the same as with `cron` and `legion-cron`, but the difference is that uWSGI will keep track of execution state and not execute the cronjob again until it is complete.

Example:

```
[uwsgi]
cron = -1 -1 -1 -1 -1 sleep 70
```

This would execute `sleep 70` every minute, but sleep command will be running longer than our execution interval, we will end up with a growing number of sleep processes. To fix this we can simply replace `cron` with `unique-cron` and uWSGI will make sure that only single sleep process is running. A new process will be started right after the previous one finishes.

Harakiri

Note: Available since 1.9.11.

`--cron-harakiri` will enforce a time limit on executed commands. If any command is taking longer it will be killed.

```
[uwsgi]

cron = sleep 30
cron-harakiri = 10
```

This will kill the cron command after 10 seconds. Note that `cron-harakiri` is a global limit, it affects all cron commands. To set a per-command time limit, use the `cron2` option (see below).

New syntax for cron options

Note: Available since 1.9.11

To allow better control over crons, a new option was added to uWSGI:

```
[uwsgi]
cron2 = option1=value,option2=value command to execute
```

Example:

```
[uwsgi]

cron2 = minute=-2,unique=1 sleep 130
```

Will spawn an unique cron command `sleep 130` every 2 minutes.

Option list is optional, available options for every cron:

- `minute` - minute part of crontab entry, default is -1 (interpreted as *)
- `hour` - hour part of crontab entry, default is -1 (interpreted as *)
- `day` - day part of crontab entry, default is -1 (interpreted as *)
- `month` - month part of crontab entry, default is -1 (interpreted as *)
- `week` - week part of crontab entry, default is -1 (interpreted as *)
- `unique` - marks cron command as unique (see above), default is 0 (not unique)
- `harakiri` - set harakiri timeout (in seconds) for this cron command, default is 0 (no harakiri)
- `legion` - set legion name for use with this cron command, cron legions are only executed on the legion lord node.

Note that you cannot use spaces in options list. (`minute=1, hour=2` will not work, but `minute=1,hour=2` will work just fine.) If any option is missing, a default value is used.

```
[uwsgi]
# execute ''my command'' every minute (-1 -1 -1 -1 -1 crontab).
cron2 = my command
# execute unique command ''/usr/local/bin/backup.sh'' at 5:30 every day.
cron2 = minute=30,hour=5,unique=1 /usr/local/bin/backup.sh
```

```
[uwsgi]
legion = mycluster 225.1.1.1:1717 100 bf-cbc:hello
legion-node = mycluster 225.1.1.1:1717
cron2 = minute=-10,legion=mycluster my command
```

This will disable harakiri for `my command`, but other cron commands will still be killed after 10 seconds:

```
[uwsgi]
cron-harakiri = 10
cron2 = harakiri=0 my command
cron2 = my second command
```

6.5 The uWSGI FastRouter

For advanced setups uWSGI includes the "fastrouter" plugin, a proxy/load-balancer/router speaking the uwsgi protocol. It is built in by default. You can put it between your webserver and real uWSGI instances to have more control over the routing of HTTP requests to your application servers.

6.5.1 Getting started

First of all you have to run the fastrouter, binding it to a specific address. Multiple addresses are supportd as well.

```
uwsgi --fastrouter 127.0.0.1:3017 --fastrouter /tmp/uwsgi.sock --fastrouter @foobar
```

Note: This is the most useless Fastrouter setup in the world.

Congratulations! You have just run the most useless Fastrouter setup in the world. Simply binding the fastrouter to a couple of addresses will not instruct it on how to route requests. To give it intelligence you have to tell it how to route requests.

6.5.2 Way 1: –fastrouter-use-base

This option will tell the fastrouter to connect to a UNIX socket with the same name of the requested host in a specified directory.

```
uwsgi --fastrouter 127.0.0.1:3017 --fastrouter-use-base /tmp/sockets
```

If you receive a request for `example.com` the fastrouter will forward the request to `/tmp/sockets/example.com`.

6.5.3 Way 2: –fastrouter-use-pattern

Same as the previous setup but you will be able to use a pattern, with `%s` mapping to the requested key/hostname.

```
uwsgi --fastrouter 127.0.0.1:3017 --fastrouter-use-base /tmp/sockets/%s/uwsgi.sock
```

Requests for `example.com` will be mapped to `/tmp/sockets/example.com/uwsgi.sock`.

6.5.4 Way 3: –fastrouter-use-cache

You can store the key/value mappings in the *uWSGI cache*. Choose a way to fill the cache, for instance a Python script like this...

```
import uwsgi
# Requests for example.com on port 8000 will go to 127.0.0.1:4040
uwsgi.cache_set("example.com:8000", "127.0.0.1:4040")
# Requests for unbit.it will go to 127.0.0.1:4040 with the modifier1 set to 5 (perl/PSGI)
uwsgi.cache_set("unbit.it", "127.0.0.1:4040,5")
```

Then run your Fastrouter-enabled server, telling it to run the script first.

```
uwsgi --fastrouter 127.0.0.1:3017 --fastrouter-use-cache --cache 100 --file foobar.py
```

6.5.5 Way 4: –fastrouter-subscription-server

This is probably one of the best way for massive auto-scaling hosting. It uses the *subscription server* to allow instances to announce themselves and subscribe to the fastrouter.

```
uwsgi --fastrouter 127.0.0.1:3017 --fastrouter-subscription-server 192.168.0.100:7000
```

This will spawn a subscription server on address 192.168.0.100 port 7000

Now you can spawn your instances subscribing to the fastrouter:

```
uwsgi --socket :3031 -M --subscribe-to 192.168.0.100:7000:example.com
uwsgi --socket :3032 -M --subscribe-to 192.168.0.100:7000:unbit.it,5 --subscribe-to 192.168.0.100:700
```

As you probably noted, you can subscribe to multiple fastrouters, with multiple keys. Multiple instances subscribing to the same fastrouter with the same key will automatically get load balanced and monitored. Handy, isn't it? Like with the caching key/value store, `modifier1` can be set with a comma. (`,5` above) Another feature of the subscription system is avoiding to choose ports. You can bind instances to random port and the subscription system will send the real value to the subscription server.

```
uwsgi --socket 192.168.0.100:0 -M --subscribe-to 192.168.0.100:7000:example.com
```

Mapping files

If you need to specify a massive amount of keys, you can use a mapping file instead.

```
# mappings.txt
unbit.it
unbit.it:8000,5
uwsgi.it
projects.unbit.it

uwsgi --socket :3031 -M --subscribe-to 192.168.0.100:7000:@mappings.txt
```

6.5.6 Way 5: –fastrouter-use-code-string

If Darth Vader wears a t-shirt with your face (and in some other corner cases too), you can customize the fastrouter with code-driven mappings. Choose a uWSGI-supported language (like Python or Lua) and define your mapping function.

```
def get(key):
    return '127.0.0.1:3031'

uwsgi --fastrouter 127.0.0.1:3017 --fastrouter-use-code-string 0:mapper.py:get
```

This will instruct the fastrouter to load the script `mapper.py` using plugin (modifier1) 0 and call the 'get' global, passing it the key. In the previous example you will always route requests to 127.0.0.1:3031. Let's create a more advanced system, for fun!

```
domains = {}
domains['example.com'] = {'nodes': ('127.0.0.1:3031', '192.168.0.100:3032'), 'node': 0}
domains['unbit.it'] = {'nodes': ('127.0.0.1:3035,5', '192.168.0.100:3035,5'), 'node': 0}

DEFAULT_NODE = '192.168.0.1:1717'

def get(key):
```

```
    if key not in domains:
        return DEFAULT_NODE

    # get the node to forward requests to
    nodes = domains[key]['nodes']
    current_node = domains[key]['node']
    value = nodes[current_node]

    # round robin :P
    next_node = current_node + 1
    if next_node >= len(nodes):
        next_node = 0

    domains[key]['node'] = next_node

    return value
```

```
uwsgi --fastrouter 127.0.0.1:3017 --fastrouter-use-code-string 0:megamapper.py:get
```

With only few lines we have implemented round-robin load-balancing with a fallback node. Pow! You could add some form of node monitoring, starting threads in the script, or other insane things. (Be sure to add them to the docs!)

Attention: Remember to not put blocking code in your functions. The fastrouter is totally non-blocking, do not ruin it!

6.5.7 Cheap mode and shared sockets

A common setup is having a webserver/proxy connected to a fastrouter and a series of uWSGI instances subscribed to it. Normally you'd use the webserver node as a uWSGI instance node. This node will subscribe to the local fastrouter. Well... don't waste cycles on that! Shared sockets are a way to share sockets among various uWSGI components. Let's use that to share a socket between the fastrouter and uWSGI instance.

```
[uwsgi]
;create a shared socket (the webserver will connect to it)
shared-socket = 127.0.0.1:3031

; bind the fastrouter to the shared socket
fastrouter = =0
; bind an instance to the same socket
socket = =0

; having a master is always a good thing...
master = true
; our subscription server
fastrouter-subscription-server = 192.168.0.100:4040
; our app
wsgi-file = /var/www/myheavyapp.wsgi
; a bunch of processes
processes = 4
; and put the fastrouter in cheap mode
fastrouter-cheap = true
```

With this setup your requests will go directly to your app (no proxy overhead) or to the fastrouter (to pass requests to remote nodes). When the fastrouter is in cheap mode, it will not respond to requests until a node is available. This means that when there are no nodes subscribed, only your local app will respond. When all of the nodes go down, the fastrouter will return in cheap mode. Seeing a pattern? Another step to awesome autoscaling.

6.5.8 Notes

- The fastrouter uses the following vars (in order of precedence) to choose a key to use:
  - `UWSGI_FASTROUTER_KEY` - the most versatile, as it doesn't depend on the request in any way
  - `HTTP_HOST`
  - `SERVER_NAME`
- You can increase the number of async events the fastrouter can manage (by default it is system-dependent) using –fastrouter-events

You can change the default timeout with –fastrouter-timeout By default the fastrouter will set fd socket passing when used over unix sockets. If you do not want it add –no-fd-passing

6.6 uWSGI internal routing

`Updated to 1.9`

As of uWSGI 1.9, a programmable internal routing subsystem is available (older releases after 1.1 have a less featureful version). You can use the internal routing subsystem to dynamically alter the way requests are handled. For example you can use it to trigger a 301 redirect on specific URLs, or to serve content from the cache on specific conditions. The internal routing subsystem is inspired by Apache's `mod_rewrite` and Linux's `iptables` command. Please, before blasting it for being messy, not-elegant nor Turing-complete, remember that it must be FAST and only FAST. If you need elegance and more complexity, do that in your code.

6.6.1 The routing chains

During the request cycle, various "chains" are traversed. Each chain contains a routing table (see below).

Chains can be "recursive". A "recursive" chain can be called multiple times in a request cycle.

This is the order of chains:

`request` it is applied before the request is passed to the plugin

`error` it is applied as soon as an HTTP status code is generate (recursive chain)

`response` it is is applied after the last response header has been generated (just before sending the body)

`final` it is aplied after the response has been sent to the client

The `request` chain is (for convention) the 'default' one, so its options are not prefixed, while the others requires a prefix.

Example:

`route-user-agent` -> happens in the request chain

while

`response-route-uri` -> happens in the response chain

6.6.2 The internal routing table

The internal routing table is a sequence of ''rules" executed one after another (forward jumps are allowed too). Each rule is composed by a ''subject'', a ''condition" and an ''action" The ''condition" is generally a PCRE regexp applied

to the subject, if it matches the action is triggered. Subjects are request's variables. Currently the following subjects are supported:

- `host` (check HTTP_HOST)
- `uri` (check REQUEST_URI)
- `qs` (check QUERY_STRING)
- `remote-addr` (check REMOTE_ADDR)
- `remote-user` (check REMOTE_USER)
- `referer` (check HTTP_REFERER)
- `user-agent` (check HTTP_USER_AGENT)
- `status` (check HTTP response status code, not available in the request chain)
- `default` (default subject, maps to PATH_INFO)

In addition to this, a pluggable system of lower-level conditions is available. You can access this system using the `--route-if` option. Currently the following checks are supported:

- `exists` (check if the subject exists in the filesystem)
- `isfile` (check if the subject is a file)
- `isdir` (check if the subject is a directory)
- `isexec` (check if the subject is an executable file)
- `equal`/`isequal`/`eq`/== (check if the subject is equal to the specified pattern)
- `ishigherequal`/>=
- `ishigher`/>
- `islower`/<
- `islowerequal`/<=
- `startswith` (check if the subject starts with the specified pattern)
- `endswith` (check if the subject ends with the specified pattern)
- `regexp`/re (check if the subject matches the specified regexp)
- `empty` (check if the subject is empty)
- `contains`

When a check requires a pattern (like with 'equal' or 'regexp') you split it from the subject with a semicolon:

```
; never matches
route-if = equal:FOO;BAR log:never here
; matches
route if = regexp:FOO;^F log:starts with F
```

Actions are the functions to run if a rule matches. This actions are exported by plugins and have a return value.

6.6.3 Action return values

Each action has a return value which tells the routing engine what to do next. The following return codes are supported:

- `NEXT` (continue to the next rule)

- CONTINUE (stop scanning the internal routing table and run the request)
- BREAK (stop scanning the internal routing table and close the request)
- GOTO x (go to rule x)

When a rule does not match, NEXT is assumed.

6.6.4 The first example

```
[uwsgi]
route-user-agent = .*curl.* redirect:http://uwsgi.it
route-remote-addr = ^127\.0\.0\.1$ break:403 Forbidden
route = ^/test log:someone called /test
route = \.php$ rewrite:/index.php
route = .* addheader:Server: my uWSGI server
route-host = ^localhost$ logvar:local=1
route-uri = ^/foo/(.*)\.jpg$ cache:key=$1.jpg
route-if = equal:${PATH_INFO};/bad break:500 Internal Server Error
```

The previous rules, build the following table:

- if the HTTP_USER_AGENT var contains 'curl' redirect the request to http://uwsgi.it (code 302, action returns BREAK)
- if REMOTE_ADDR is '127.0.0.1' returns a 403 Forbidden (action returns BREAK)
- if PATH_INFO starts with /test print the string 'someone called /test' in the logs (action returns NEXT)
- if PATH_INFO ends with '.php' rewrite it to /index.php (action returns NEXT)
- for all of the PATH_INFO add the HTTP header 'Server: my uWSGI server' to the response (action returns NEXT)
- if HTTP_HOST is localhost add the logvar 'local' setting it to '1'
- if REQUEST_URI starts with /foo and ends with .jpg get it from the uWSGI cache using the supplied key (built over regexp grouping) (action returns BREAK)
- if the PATH_INFO is equal to /bad throws a 500 error

6.6.5 Accessing request vars

In addition to PCRE placeholders/groups (using $1 to $9) you can access request variables (PATH_INFO, SCRIPT_NAME, REQUEST_METHOD...) using the ${VAR} syntax.

```
[uwsgi]
route-user-agent = .*curl.* redirect:http://uwsgi.it${REQUEST_URI}
route-remote-addr = ^127\.0\.0\.1$ break:403 Forbidden for ip ${REMOTE_ADDR}
```

6.6.6 Accessing cookies

You can access a cookie value using the ${cookie[name]} syntax:

```
[uwsgi]
route = ^/foo log:${cookie[foobar]}
```

this will log the content of the 'foobar' cookie of the current request

6.6.7 Accessing query string items

You can access the value of the HTTP query string using the ${qs[name]} syntax:

```
[uwsgi]
route = ^/foo log:${qs[foobar]}
```

this will log the content of the 'foobar' item of the current request's query string

6.6.8 Pluggable routing variables

Both the cookie and qs vars, are so-called "routing vars". They are pluggable, so external plugins can add new vars to add new features to your application. (Check the *The GeoIP plugin* plugin for an example of this.) A number of embedded routing variables are also available.

- `mime` – returns the mime type of the specified var: ${mime[REQUEST_URI]}

  ```
  [uwsgi]
  route = ^/images/(.+) addvar:MYFILE=$1.jpg
  route = ^/images/ addheader:Content-Type: ${mime[MYFILE]}
  ```

- `time` – returns time/date in various form. The only supported (for now) is time[unix] returning the epoch
- `math` – requires matheval support. Example: math[CONTENT_LENGTH+1]
- `base64` – encode the specified var in base64
- `hex` – encode the specified var in hex
- `uwsgi` – return internal uWSGI information, uwsgi[wid], uwsgi[pid], uwsgi[uuid] and uwsgi[status] are currently supported

6.6.9 Is –route-if not enough? Why –route-uri and friends?

This is a good question. You just need to always remember that uWSGI is about versatility and *performance*. Gaining cycles is always good. The `--route-if` option, while versatile, cannot be optimized as all of its parts have to be recomputed at every request. This is obviously very fast, but `--route-uri` option (and friends) can be pre-optimized (during startup) to directly map to the request memory areas, so if you can use them, you definitely should. :)

6.6.10 GOTO

Yes, the most controversial construct of the whole information technology industry (and history) is here. You can make forward (only forward!) jumps to specific points of the internal routing table. You can set labels to mark specific point of the table, or if you are brave (or foolish) jump directly to a rule number. Rule numbers are printed on server startup, but please use labels.

```
[uwsgi]

route-host = ^localhost$ goto:localhost
route-host = ^sid\.local$ goto:sid.local
route = .* last:

route-label = sid.local
route-user-agent = .*curl.* redirect:http://uwsgi.it
route-remote-addr = ^192\.168\..* break:403 Forbidden
```

```
route = ^/test log:someone called /test
route = \.php$ rewrite:/index.php
route = .* addheader:Server: my sid.local server
route = .* logvar:local=0
route-uri = ^/foo/(.*)\.jpg$ cache:key=$1.jpg
route = .* last:

route-label = localhost
route-user-agent = .*curl.* redirect:http://uwsgi.it
route-remote-addr = ^127\.0\.0\.1$ break:403 Forbidden
route = ^/test log:someone called /test
route = \.php$ rewrite:/index.php
route = .* addheader:Server: my uWSGI server
route = .* logvar:local=1
route-uri = ^/foo/(.*)\.jpg$ cache:key=$1.jpg
route = .* last:
```

The example is like the previous one, but we with some differences between domains. Check the use of "last:", to interrupt the routing table scan. You can rewrite the first 2 rules as one:

```
[uwsgi]

route-host = (.*) goto:$1
```

6.6.11 Collecting response headers

As we have already seen, each uWSGI request has a set of variables associated. They are generally the CGI vars passed by the webserver, but you can extend them with other variables too (check the 'addvar' action).

uWSGI 1.9.16 added a new feature allowing you to store the content of a response header in a request var. This simplify the write of more advanced rules.

For example you may want to gzip all of the text/html responses:

```
[uwsgi]
; store Content-Type response header in MY_CONTENT_TYPE var
collect-header = Content-Type MY_CONTENT_TYPE
; if response is text/html, and client supports it, gzip it
response-route-if = equal:${MY_CONTENT_TYPE};text/html goto:gzipme
response-route-run = last:

response-route-label = gzipme
; gzip only if the client support it
response-route-if = contains:${HTTP_ACCEPT_ENCODING};gzip gzip:
```

6.6.12 The available actions

continue/last

Return value: `CONTINUE`

Stop the scanning of the internal routing table and continue to the selected request handler.

break

Return value: `BREAK`

Stop scanning the internal routing table and close the request. Can optionally returns the specified HTTP status code:

```
[uwsgi]
route = ^/notfound break:404 Not Found
route = ^/bad break:
route = ^/error break:500
```

log

Return value: `NEXT`

Print the specified message in the logs.

```
[uwsgi]
route = ^/logme/(.) log:hey i am printing $1
```

logvar

Return value: `NEXT`

Add the specified logvar.

```
[uwsgi]
route = ^/logme/(.) logvar:item=$1
```

goto

Return value: `NEXT`

Make a forward jump to the specified label or rule position

addvar

Return value: `NEXT`

Add the specified CGI (environment) variable to the request.

```
[uwsgi]
route = ^/foo/(.) addvar:FOOVAR=prefix$1suffix
```

addheader

Return value: `NEXT`

Add the specified HTTP header to the response.

```
[uwsgi]
route = ^/foo/(.) addheader:Foo: Bar
```

delheader//remheader

Return value: NEXT

Remove the specified HTTP header from the response.

```
[uwsgi]
route = ^/foo/(.) delheader:Foo
```

signal

Return value: NEXT

Raise the specified uwsgi signal.

send

Return value: NEXT

Extremely advanced (and dangerous) function allowing you to add raw data to the response.

```
[uwsgi]
route = ^/foo/(.) send:destroy the world
```

send-crnl

Return value: NEXT

Extremely advanced (and dangerous) function allowing you to add raw data to the response, suffixed with rn.

```
[uwsgi]
route = ^/foo/(.) send-crnl:HTTP/1.0 100 Continue
```

redirect/redirect-302

Return value: BREAK

Plugin: router_redirect

Return a HTTP 302 Redirect to the specified URL.

redirect-permanent/redirect-301

Return value: BREAK

Plugin: router_redirect

Return a HTTP 301 Permanent Redirect to the specified URL.

rewrite

Return value: `NEXT`

Plugin: `router_rewrite`

A rewriting engine inspired by Apache mod_rewrite. Rebuild PATH_INFO and QUERY_STRING according to the specified rules before the request is dispatched to the request handler.

```
[uwsgi]
route-uri = ^/foo/(.*) rewrite:/index.php?page=$1.php
```

rewrite-last

Alias for rewrite but with a return value of `CONTINUE`, directly passing the request to the request handler next.

uwsgi

Return value: `BREAK`

Plugin: `router_uwsgi`

Rewrite the modifier1, modifier2 and optionally `UWSGI_APPID` values of a request or route the request to an external uwsgi server.

```
[uwsgi]
route = ^/psgi uwsgi:127.0.0.1:3031,5,0
```

This configuration routes all of the requests starting with `/psgi` to the uwsgi server running on 127.0.0.1:3031 setting modifier1 to 5 and modifier2 to 0. If you only want to change the modifiers without routing the request to an external server, use the following syntax.

```
[uwsgi]
route = ^/psgi uwsgi:,5,0
```

To set a specific `UWSGI_APPID` value, append it.

```
[uwsgi]
route = ^/psgi uwsgi:127.0.0.1:3031,5,0,fooapp
```

The subrequest is async-friendly (engines such as gevent or ugreen are supported) and if offload threads are available they will be used.

http

Return value: `BREAK`

Plugin: `router_http`

Route the request to an external HTTP server.

```
[uwsgi]
route = ^/zope http:127.0.0.1:8181
```

You can substitute an alternative Host header with the following syntax:

```
[uwsgi]
route = ^/zope http:127.0.0.1:8181,myzope.uwsgi.it
```

static

Return value: `BREAK`

Plugin: `router_static`

Serve a static file from the specified physical path.

```
[uwsgi]
route = ^/logo static:/var/www/logo.png
```

basicauth

Return value: `NEXT` or `BREAK 401` on failed authentication

Plugin: `router_basicauth`

Four syntaxes are supported.

- `basicauth:realm,user:password` – a simple user:password mapping
- `basicauth:realm,user:` – only authenticates username
- `basicauth:realm,htpasswd` – use a htpasswd-like file. All POSIX crypt() algorithms are supported. This is _not_ the same behavior as Apache's traditional htpasswd files, so use the `-d` flag of the htpasswd utility to create compatible files.
- `basicauth:realm,` – Useful to cause a HTTP 401 response immediately. As routes are parsed top-bottom, you may want to raise that to avoid bypassing rules.

Example:

```
[uwsgi]
route = ^/foo basicauth-next:My Realm,foo:bar
route = ^/foo basicauth:My Realm,foo2:bar2
route = ^/bar basicauth:Another Realm,kratos:
```

Example: using basicauth for Trac

```
[uwsgi]
; load plugins (if required)
plugins = python,router_basicauth

; bind to port 9090 using http protocol
http-socket = :9090

; set trac instance path
env = TRAC_ENV=myinstance
; load trac
module = trac.web.main:dispatch_request

; trigger authentication on /login
route = ^/login basicauth-next:Trac Realm,pippo:pluto
route = ^/login basicauth:Trac Realm,foo:bar

;high performance file serving
static-map = /chrome/common=/usr/local/lib/python2.7/dist-packages/trac/htdocs
```

basicauth-next

same as `basicauth` but returns `NEXT` on failed authentication.

ldapauth

Return value: `NEXT` or `BREAK 401` on failed authentication

Plugin: `ldap`

This auth router is part of the LDAP plugin, so it has to be loaded in order for this to be available. It's like the basicauth router, but uses an LDAP server for authentication, syntax: `ldapauth:realm,options` Available options:

- `url` - LDAP server URI (required)
- `binddn` - DN used for binding. Required if the LDAP server does not allow anonymous searches.
- `bindpw` - password for the `binddn` user.
- `basedn` - base DN used when searching for users (required)
- `filter` - filter used when searching for users (default is "(objectClass=*)")
- `attr` - LDAP attribute that holds user login (default is "uid")
- `loglevel` - 0 - don't log any binds, 1 - log authentication errors, 2 - log both successful and failed binds

Example:

```
route = ^/protected ldapauth:LDAP auth realm,url=ldap://ldap.domain.com;basedn=ou=users,dc=domain;bi
```

ldapauth-next

Same as ldapauth but returns `NEXT` on failed authentication.

cache

Return value: `BREAK`

Plugin: `router_cache`

cachestore/cache-store

cachevar

cacheset

memcached

rpc

The "rpc" routing instruction allows you to call uWSGI RPC functions directly from the routing subsystem and forward their output to the client.

```
[uwsgi]
http-socket = :9090
route = ^/foo addheader:Content-Type: text/html
route = ^/foo rpc:hello ${REQUEST_URI} ${HTTP_USER_AGENT}
route = ^/bar/(.+)$ rpc:test $1 ${REMOTE_ADDR} uWSGI %V
route = ^/pippo/(.+)$ rpc:test@127.0.0.1:4141 $1 ${REMOTE_ADDR} uWSGI %V
import = funcs.py
```

call

Plugin: `rpc`

rpcret

Plugin: `rpc`

rpcret calls the specified rpc function and uses its return value as the action return code (next, continue, goto, etc)

rpcblob//rpcnext

Plugin: `rpc`

rpcnext/rpcblob calls the specified RPC function, sends the response to the client and continues to the next rule.

rpcraw

Plugin: `rpc`

access

spnego

In development...

radius

In development...

xslt

See also:

The XSLT plugin

ssi

See also:

SSI (Server Side Includes) plugin

gridfs

See also:

The GridFS plugin

donotlog

chdir

setapp

setuser

sethome

setfile

setscriptname

setprocname

alarm

flush

fixcl

cgi

Plugin: `cgi`

cgihelper

Plugin: `cgi`

access

Plugin: `router_access`

cache-continue

Plugin: `router_cache`

cachevar

Plugin: `router_cache`

cacheinc

Plugin: router_cache

cachedec

Plugin: router_cache

cachemul

Plugin: router_cache

cachediv

Plugin: router_cache

proxyhttp

Plugin: router_http

memcached

Plugin: router_memcached

memcached-continue

Plugin: router_memcached

memcachedstore

Plugin: router_memcached

memcached-store

Plugin: router_memcached

redis

Plugin: router_redis

redis-continue

Plugin: router_redis

redisstore

Plugin: router_redis

redis-store

Plugin: `router_redis`

proxyuwsgi

Plugin: `router_uwsgi`

harakiri

Set harakiri for the current request.

file

Directly transfer the specified filename *without* using acceleration (sendfile, offloading, etc.).

```
[uwsgi]
http-socket = :9090
route-run = file:filename=/var/www/${PATH_INFO}
```

clearheaders

clear the response headers, setting a new HTTP status code, useful for resetting a response

```
[uwsgi]
http-socket = :9090
response-route = ^/foo goto:foobar
response-route-run = last:

response-route-label = foobar
response-route-run = clearheaders:404 Not Found
response-route-run = addheader:Content-Type: text/html
```

resetheaders

alias for clearheaders

6.7 The uWSGI Legion subsystem

As of uWSGI 1.9-dev a new subsystem for clustering has been added: The Legion subsystem. A Legion is a group of uWSGI nodes constantly fighting for domination. Each node has a valor value (different from the others, if possible). The node with the highest valor is the Lord of the Legion (or if you like a less gaming nerd, more engineer-friendly term: the master). This constant fight generates 7 kinds of events:

1. `setup` - when the legion subsystem is started on a node
2. `join` - the first time quorum is reached, only on the newly joined node
3. `lord` - when this node becomes the lord
4. `unlord` - when this node loses the lord title

5. `death` - when the legion subsystem is shutting down
6. `node-joined` - when any new node joins our legion
7. `node-left` - when any node leaves our legion

You can trigger actions every time such an event rises.

Note: `openssl` headers must be installed to build uWSGI with Legion support.

6.7.1 IP takeover

This is a very common configuration for clustered environments. The IP address is a resource that must be owned by only one node. For this example, that node is our Lord. If we configure a Legion right (remember, a single uWSGI instances can be a member of all of the legions you need) we could easily implement IP takeover.

```
[uwsgi]

legion = clusterip 225.1.1.1:4242 98 bf-cbc:hello
legion-node = clusterip 225.1.1.1:4242

legion-lord = clusterip cmd:ip addr add 192.168.173.111/24 dev eth0
legion-lord = clusterip cmd:arping -c 3 -S 192.168.173.111 192.168.173.1

legion-setup = clusterip cmd:ip addr del 192.168.173.111/24 dev eth0
legion-unlord = clusterip cmd:ip addr del 192.168.173.111/24 dev eth0
legion-death = clusterip cmd:ip addr del 192.168.173.111/24 dev eth0
```

In this example we join a legion named `clusterip`. To receive messages from the other nodes we bind on the multicast address 225.1.1.1:4242. The valor of this node will be 98 and each message will be encrypted using Blowfish in CBC with the shared secret `hello`. The `legion-node` option specifies the destination of our announce messages. As we are using multicast we only need to specify a single "node". The last options are the actions to trigger on the various states of the cluster. For an IP takeover solution we simply rely on the Linux `iproute` commands to set/unset ip addresses and to send an extra ARP message to announce the change. Obviously this specific example requires root privileges or the `CAP_NET_ADMIN` Linux capability, so be sure to not run untrusted applications on the same uWSGI instance managing IP takeover.

The Quorum

To choose a Lord each member of the legion has to cast a vote. When all of the active members of a legion agree on a Lord, the Lord is elected and the old Lord is demoted. Every time a new node joins or leaves a legion the quorum is re-computed and logged to the whole cluster.

Choosing the Lord

Generally the node with the higher valor is chosen as the Lord, but there can be cases where multiple nodes have the same valor. When a node is started a UUID is assigned to it. If two nodes with same valor are found the one with the lexicographically higher UUID wins.

Split brain

Even though each member of the Legion has to send a checksum of its internal cluster-membership, the system is still vulnerable to the split brain problem. If a node loses network connectivity with the cluster, it could believe it is the only node available and starts going in Lord mode.

For many scenarios this is not optimal. If you have more than 2 nodes in a legion you may want to consider tuning the quorum level. The quorum level is the amount of votes (as opposed to nodes) needed to elect a lord. `legion-quorum` is the option for the job. You can reduce the split brain problem asking the Legion subsystem to check for at least 2 votes:

```
[uwsgi]

legion = clusterip 225.1.1.1:4242 98 bf-cbc:hello
legion-node = clusterip 225.1.1.1:4242

legion-quorum = clusterip 2

legion-lord = clusterip cmd:ip addr add 192.168.173.111/24 dev eth0
legion-lord = clusterip cmd:arping -c 3 -S 192.168.173.111 192.168.173.1

legion-setup = clusterip cmd:ip addr del 192.168.173.111/24 dev eth0
legion-unlord = clusterip cmd:ip addr del 192.168.173.111/24 dev eth0
legion-death = clusterip cmd:ip addr del 192.168.173.111/24 dev eth0
```

As of 1.9.7 you can use nodes with valor 0 (concept similar to MongoDB's Arbiter Nodes), such nodes will be counted when checking for quorum but may never become The Lord. This is useful when you only need a couple nodes while protecting against split-brain.

Actions

Each one of the four phases of a legion can trigger an action. The actions system is modular so you can add new kinds of actions. Currently the supported actions are:

6.7.2 `cmd:<command>`

Run a shell command.

6.7.3 `signal:<num>`

Raise a uWSGI signal.

6.7.4 `log:<msg>`

Log a message. For example you could combine the log action with the alarm subsystem to have cluster monitoring for free.

Multicast, broadcast and unicast

Even if multicast is probably the easiest way to implement clustering it is not available in all networks. If multicast is not an option, you can rely on normal IP addresses. Just bind to an address and add all of the legion-node options you need:

```
[uwsgi]

legion = mycluster 192.168.173.17:4242 98 bf-cbc:hello
legion-node = mycluster 192.168.173.22:4242
```

```
legion-node = mycluster 192.168.173.30:4242
legion-node = mycluster 192.168.173.5:4242
```

This is for a cluster of 4 nodes (this node + 3 other nodes)

Multiple Legions

You can join multiple legions in the same instance. Just remember to use different addresses (ports in case of multicast) for each legion.

```
[uwsgi]

legion = mycluster 192.168.173.17:4242 98 bf-cbc:hello
legion-node = mycluster 192.168.173.22:4242
legion-node = mycluster 192.168.173.30:4242
legion-node = mycluster 192.168.173.5:4242

legion = mycluster2 225.1.1.1:4243 99 aes-128-cbc:secret
legion-node = mycluster2 225.1.1.1:4243

legion = anothercluster 225.1.1.1:4244 91 aes-256-cbc:secret2
legion-node = anothercluster 225.1.1.1:4244
```

Security

Each packet sent by the Legion subsystem is encrypted using a specified cipher, a preshared secret, and an optional IV (initialization vector). Depending on cipher, the IV may be a required parameter. To get the list of supported ciphers, run `openssl enc -h`.

Important: Each node of a Legion has to use the same encryption parameters.

To specify the IV just add another parameter to the **legion** option.

```
[uwsgi]

legion = mycluster 192.168.173.17:4242 98 bf-cbc:hello thisistheiv
legion-node = mycluster 192.168.173.22:4242
legion-node = mycluster 192.168.173.30:4242
legion-node = mycluster 192.168.173.5:4242
```

To reduce the impact of replay-based attacks, packets with a timestamp lower than 30 seconds are rejected. This is a tunable parameter. If you have no control on the time of all of the nodes you can increase the clock skew tolerance.

Tuning and Clock Skew

Currently there are three parameters you can tune. These tuables affect all Legions in the system. The frequency (in seconds) at which each packet is sent (**legion-freq <secs>**), the amount of seconds after a node not sending packets is considered dead (**legion-tolerance <secs>**), and the amount of clock skew between nodes (**legion-skew-tolerance <secs>**). The Legion subsystem requires tight time synchronization, so the use of NTP or similar is highly recommended. By default each packet is sent every 3 seconds, a node is considered dead after 15 seconds, and a clock skew of 30 seconds is tolerated. Decreasing skew tolerance should increase security against replay attacks.

Lord scroll (coming soon)

The Legion subsystem can be used for a variety of purposes ranging from master election to node autodiscovery or simple monitoring. One example is to assign a "blob of data" (a scroll) to every node, One use of this is to pass reconfiguration parameters to your app, or to log specific messages. Currently the scroll system is being improved upon, so if you have ideas join our mailing list or IRC channel.

Legion API

You can know if the instance is a lord of a Legion by simply calling

```
int uwsgi_legion_i_am_the_lord(char *legion_name);
```

It returns 1 if the current instance is the lord for the specified Legion.

- The Python plugin exposes it as `uwsgi.i_am_the_lord(name)`
- The PSGI plugin exposes it as `uwsgi::i_am_the_lord(name)`
- The Rack plugin exposes it as `UWSGI::i_am_the_lord(name)`

Obviously more API functions will be added in the future, feel free to expose your ideas.

Stats

The Legion information is available in the *The uWSGI Stats Server*. Be sure to understand the difference between "nodes" and "members". Nodes are the peer you configure with the **legion-node** option while members are the effective nodes that joined the cluster.

The old clustering subsystem

During 0.9 development cycle a clustering subsystem (based on multicast) was added. It was very raw, unreliable and very probably no-one used it seriously. The new method is transforming it in a general API that can use different backends. The Legion subsystem can be one of those backends, as well as projects like corosync or the redhat cluster suite.

6.8 Locks

uWSGI supports a configurable number of locks you can use to synchronize worker processes. Lock 0 (zero) is always available, but you can add more with the `locks` option. If your app has a lot of critical areas, holding and releasing the same lock over and over again can kill performance.

```
def use_lock_zero_for_important_things():
    uwsgi.lock() # Implicit parameter 0
    # Critical section
    uwsgi.unlock() # Implicit parameter 0

def use_another_lock():
    uwsgi.lock(1)
    time.sleep(1) # Take that, performance! Ha!
    uwsgi.unlock(1)
```

6.9 uWSGI Mules

Mules are worker processes living in the uWSGI stack but not reachable via socket connections, that can be used as a generic subsystem to offload tasks. You can see them as a more primitive *spooler*.

They can access the entire uWSGI API and can manage signals and be communicated with through a simple string-based message system.

To start a mule (you can start an unlimited number of them), use the `mule` option as many times as you need.

Mules have two modes,

- Signal only mode (the default). In this mode the mules load your application as normal workers would. They can only respond to *uWSGI signals*.
- Programmed mode. In this mode mules load a program separate from your application. See ProgrammedMules.

By default each mule starts in signal-only mode.

```
uwsgi --socket :3031 --mule --mule --mule --mule
```

```
<uwsgi>
    <socket>:3031</socket>
    <mule/>
    <mule/>
    <mule/>
    <mule/>
</uwsgi>
```

6.9.1 Basic usage

```
import uwsgi
from uwsgidecorators import timer, signal, filemon

# run a timer in the first available mule
@timer(30, target='mule')
def hello(signum):
    print "Hi! I am responding to signal %d, running on mule %d" % (signum, uwsgi.mule_id())

# map signal 17 to mule 2
@signal(17, target='mule2')
def i_am_mule2(signum):
    print "Greetings! I am running in mule number two."

# monitor /tmp and arouse all of the mules on modifications
@filemon('/tmp', target='mules')
def tmp_modified(signum):
    print "/tmp has been modified. I am mule %d!" % uwsgi.mule_id()
```

6.9.2 Giving a brain to mules

As mentioned before, mules can be programmed. To give custom logic to a mule, pass the name of a script to the `mule` option.

```
uwsgi --socket :3031 --mule=somaro.py --mule --mule --mule
```

This will run 4 mules, 3 in signal-only mode and one running `somaro.py`.

```
# somaro.py
from threading import Thread
import time

def loop1():
    while True:
        print "loop1: Waiting for messages... yawn."
        message = uwsgi.mule_get_msg()
        print message

def loop2():
    print "Hi! I am loop2."
    while True:
        time.sleep(2)
        print "This is a thread!"

t = Thread(target=loop2)
t.daemon = True
t.start()

if __name__ == '__main__':
    loop1()
```

So as you can see from the example, you can use `mule_get_msg()` to receive messages in a programmed mule. Multiple threads in the same programmed mule can wait for messages.

If you want to block a mule to wait on an uWSGI signal instead of a message you can use `uwsgi.signal_wait()`.

Use `uwsgi.mule_msg()` to send a message to a programmed mule. Mule messages can be sent from anywhere in the uWSGI stack, including but not limited to workers, the spoolers, another mule.

```
# Send the string "ciuchino" to mule1.
# If you do not specify a mule ID, the message will be processed by the first available programmed mu
uwsgi.mule_msg("ciuchino", 1)
```

As you can spawn an unlimited number of mules, you may need some form of synchronization – for example if you are developing a task management subsystem and do not want two mules to be able to start the same task simultaneously. You're in luck – see *Locks*.

6.10 The uWSGI offloading subsystem

Offloading is a way to optimize tiny tasks, delegating them to one or more threads.

This threads run such tasks in a non-blocking/evented way allowing for a huge amount of concurrency.

Various component of the uWSGI stack has been made offload-friendly, and the long-term target is to allow application code to abuse it.

To start the offloading subsystem just add –offload-threads <n>, where <n> is the number of threads (per-worker) to spawn. They are native threads, they are lock-free (no shared resources), thundering-herd free (requests to the system are made in round-robin) and they are the best way to abuse your CPU cores.

The number of offloaded requests is accounted in the "offloaded_requests" metric of the stats subsystem.

6.10.1 Offloading static files

The first component made offload-aware has been the static file serving system.

When offload threads are available, the whole transfer of the file is delegated to one of those threads, freeing your worker suddenly (so it will be ready to accept new requests)

Example:

```
[uwsgi]
socket = :3031
check-static = /var/www
offload-threads = 4
```

6.10.2 Offloading internal routing

The router_uwsgi and router_http plugins are offload-friendly.

You can route requests to external uwsgi/HTTP servers without being worried about having a blocked worker during the response generation.

Example:

```
[uwsgi]
socket = :3031
offload-threads = 8
route = ^/foo http:127.0.0.1:8080
route = ^/bar http:127.0.0.1:8181
route = ^/node http:127.0.0.1:9090
```

Since 1.9.11 the `cache` router is offload friendly too.

```
[uwsgi]
socket = :3031
offload-threads = 8
route-run = cache:key=${REQUEST_URI}
```

As soon as the object is retrieved from the cache, it will be transferred in one of the offload threads.

6.10.3 The Future

The offloading subsystem has a great potential, you can think of it as a software DMA: you program it, and then it goes alone.

Currently it is pretty monolithic, but the idea is to allow more complex plugins (a redis one is in the works).

Next step is allowing the user to "program" it via the uwsgi api.

6.11 The uWSGI queue framework

In addition to the *caching framework*, uWSGI includes a shared queue.

At the low level it is a simple block-based shared array, with two optional counters, one for stack-style, LIFO usage, the other one for FIFO.

The array is circular, so when one of the two pointers reaches the end (or the beginning), it is reset. Remember this!

To enable the queue, use the `queue` option. Queue blocks are 8 KiB by default. Use `queue-blocksize` to change this.

```
# 100 slots, 8 KiB of data each
uwsgi --socket :3031 --queue 100
# 42 slots, 128 KiB of data each
uwsgi --socket :3031 --queue 42 --queue-blocksize 131072
```

6.11.1 Using the queue as a shared array

```
# Put a binary string in slot 17.
uwsgi.queue_set(17, "Hello, uWSGI queue!")

# Get it back.
print uwsgi.queue_get(17)
```

6.11.2 Using the queue as a shared stack

Warning: Remember that `uwsgi.queue_pop()` and `uwsgi.queue_last()` will remove the item or items from the queue.

```
# Push a value onto the end of the stack.
uwsgi.queue_push("Hello, uWSGI stack!")

# Pop it back
print uwsgi.queue_pop()

# Get the number of the next available slot in the stack
print uwsgi.queue_slot()

# Pop the last N items from the stack
items = uwsgi.queue_last(3)
```

6.11.3 Using the queue as a FIFO queue

Note: Currently you can only pull, not push. To enqueue an item, use `uwsgi.queue_set()`.

```
# Grab an item from the queue
uwsgi.queue_pull()
# Get the current pull/slot position (this is independent from the stack-based one)
print uwsgi.queue_pull_slot()
```

6.11.4 Notes

- You can get the queue size with `uwsgi.queue_size`.
- Use the `queue-store` option to persist the queue on disk. Use `queue-store-sync` (in master cycles – usually seconds) to force disk syncing of the queue.
- The `tests/queue.py` application is a fully working example.

6.12 uWSGI RPC Stack

uWSGI contains a fast, simple, pan-and-cross-language RPC stack.

Although you may fall in love with this subsystem, try to use it only when you need it. There are plenty of higher-level RPC technologies better suited for the vast majority of situations.

That said, the uWSGI RPC subsystem shines with its performance and memory usage. As an example, if you need to split the load of a request to multiple servers, the uWSGI RPC is a great choice, as it allows you to offload tasks with very little effort.

Its biggest limit is in its "typeless" approach.

RPC functions can take upto 254 args. Each argument has to be a string with a 16 bit max size (65535 bytes), while the return value has to be a string (this time 64bit, so no limit is in place)

Warning: 64 bit response has been implemented only in uWSGI 1.9.20, older release have 16 bit response limit

---

Note: RPC functions receive arguments in the form of binary strings, so every RPC exportable function must assume that each argument is a string. Every RPC function returns a binary string of 0 or more characters.

So, if you need "elegance" or strong typing, just look in another place (or roll your own typing on top of uWSGI RPC, maybe...).

Since 1.9 the RPC subsystem is fully async-friendly, so you can use it with gevent and Coro::AnyEvent etc.

6.12.1 Learning by example

Let's start with a simple RPC call from `10.0.0.1:3031` to `10.0.0.2:3031`.

So let's export a "hello" function on `.2`.

```
import uwsgi

def hello_world():
    return "Hello World"

uwsgi.register_rpc("hello", hello_world)
```

This uses `uwsgi.register_rpc()` to declare a function called "hello" to be exported. We'll start the server with `--socket :3031`.

On the caller's side, on `10.0.0.1`, let's declare the world's (second) simplest WSGI app.

```
import uwsgi

def application(env, start_response):
    start_response('200 Ok', [('Content-Type', 'text/html')])
    return uwsgi.rpc('10.0.0.2:3031', 'hello')
```

That's it!

What about, let's say, Lua?

Glad you asked. If you want to export functions in Lua, simply do:

```
function hello_with_args(arg1, arg2)
    return 'args are '..arg1..' '..arg2
end

uwsgi.register_rpc('hellolua', hello_with_args)
```

And in your Python WSGI app:

```
import uwsgi

def application(env, start_response):
    start_response('200 Ok', [('Content-Type', 'text/html')]
    return uwsgi.rpc('10.0.0.2:3031', 'hellolua', 'foo', 'bar')
```

And other languages/platforms?

Check the language specific docs, basically all of them should support registering and calling RPC functions.

You can build multi-languages app with really no effort at all and will be happily surprised about how easy it is to call *Java* functions from Perl, JavaScript from Python and so on.

6.12.2 Doing RPC locally

Doing RPC locally may sound a little silly, but if you need to call a Lua function from Python with the absolute least possible overhead, uWSGI RPC is your man.

If you want to call a RPC defined in the same server (governed by the same master, etc.), simply set the first parameter of `uwsgi.rpc` to None or nil, or use the convenience function `uwsgi.call()`.

6.12.3 Doing RPC from the internal routing subsystem

The RPC plugin exports a bunch of internal routing actions:

- *rpc* call the specified rpc function and send the response to the client
- *rpcnext/rpcblob* call the specified rpc function, send the response to the client and continue to the next rule
- *rpcret* calls the specified rpc function and uses its return value as the action return code (next, continue, goto ...)

```
[uwsgi]
route = ^/foo rpc:hello ${REQUEST_URI} ${REMOTE_ADDR}
; call on remote nodes
route = ^/multi rpcnext:part1@192.168.173.100:3031
route = ^/multi rpcnext:part2@192.168.173.100:3031
route = ^/multi rpcnext:part3@192.168.173.100:3031
```

6.12.4 Doing RPC from nginx

As Nginx supports low-level manipulation of the uwsgi packets sent to upstream uWSGI servers, you can do RPC directly through it. Madness!

```
location /call {
    uwsgi_modifier1 173;
    uwsgi_modifier2 1;
```

```
    uwsgi_param hellolua foo
    uwsgi_param bar ""

    uwsgi_pass 10.0.0.2:3031;

    uwsgi_pass_request_headers off;
    uwsgi_pass_request_body off;
}
```

Zero size strings will be ignored by the uWSGI array parser, so you can safely use them when the numbers of parameters + function_name is not even.

Modifier2 is set to 1 to inform that raw strings (HTTP responses in this case) are received. Otherwise the RPC subsystem would encapsulate the output in an uwsgi protocol packet, and nginx isn't smart enough to read those.

6.12.5 HTTP PATH_INFO -> RPC bridge

6.12.6 XML-RPC -> RPC bridge

6.13 SharedArea – share memory pages between uWSGI components

Warning: SharedArea is a very low-level mechanism. For an easier-to-use alternative, see the *Caching* and *Queue* frameworks.

Warning: This page refers to "new generation" sharedarea introduced in uWSGI 1.9.21, the older API is no more supported.

The sharedarea subsystem allows you to share pages of memory between your uWSGI components (workers, spoolers, mules, etc.) in a very fast (and safe) way.

Contrary to the higher-level *caching framework*, sharedarea operations are way faster (a single copy instead of the double, as required by caching) and offers various optimizations for specific needs.

Each sharedarea (yes, you can have multiple areas) has a size (generally specified in the number of pages), so if you need an 8 KiB shared area on a system with 4 KiB pages, you would use `sharedarea=2`.

The sharedarea subsystem is fully thread-safe.

6.13.1 Simple option VS keyval

The sharedarea subsystem exposes (for now) a single option: `--sharedarea`.

It takes two kinds of arguments: the number of pages (simple approach) or a keyval arg (for advanced tuning).

The following keyval keys are available:

- `pages` – set the number of pages
- `file` – create the sharedarea from a file that will be `mmap`ed
- `fd` – create the sharedarea from a file descriptor that will be `mmap`ed
- `size` – mainly useful with the `fd` and `ptr` keys to specify the size of the map (can be used as a shortcut to avoid calculation of the `pages` value too)

- `ptr` – directly map the area to the specified memory pointer.

6.13.2 The API

The API is pretty big, the sharedarea will be the de-facto toy for writing highly optimized web apps (especially for embedded systems).

Most of the documented uses make sense on systems with slow CPUs or very small amounts of memory.

`sharedarea_read(id, pos[, len])` Read `len` bytes from the specified sharedarea starting at offset `pos`. If `len` is not specified, the memory will be read til the end (starting from `pos`).

`sharedarea_write(id, pos, string)` Write the specified `string` (it is language-dependent, obviously) to the specified sharedarea at offset `pos`.

`sharedarea_read8|16|32|64(id, pos)` Read a signed integer (8, 16, 32 or 64 bit) from the specified position.

`sharedarea_write8|16|32|64(id, pos)` Write a signed integer (8, 16, 32 or 64 bit) to the specified position.

`sharedarea_inc8|16|32|64(id, pos)` Increment the signed integer (8, 16, 32 or 64 bit) at the specified position.

`sharedarea_dec8|16|32|64(id, pos)` Decrement the signed integer (8, 16, 32 or 64 bit) at the specified position.

`sharedarea_wait(id[, freq, timeout])` Wait for modifications of the specified sharedarea (see below).

`sharedarea_rlock(id)` lock a shared area for read (use only if you know what you are doing, generally the sharedarea api functions implement locking by themselves)

`sharedarea_wlock(id)` lock a shared area for write (use only if you know what you are doing, generally the sharedarea api functions implement locking by themselves)

`sharedarea_unlock(id)` unlock a shared area (use only if you know what you are doing, generally the sharedarea api functions implement locking by themselves)

6.13.3 Waiting for updates

One of the most powerful features of sharedareas (compared to caching) is "waiting for updates". Your worker/thread/async_core can be suspended until a sharedarea is modified.

Technically, a millisecond-resolution timer is triggered, constantly checking for updates (the operation is very fast, as the sharedarea object has an update counter, so we only need to check that value for changes).

6.13.4 Optional API

The following functions require specific features from the language, so not all of the language plugins are able to support them.

`sharedarea_readfast(id, pos, object, [, len])` Read `len` bytes from the specified sharedarea starting at offset `pos` to the specified object. If `len` is not specified, the memory will be read til the end (starting from `pos`). Currently is implemented only for Perl.

`sharedarea_memoryview(id)` returns python memoryview object you can directly manipulate (works only on CPython)

sharedarea_object(id) some plugin exposes an alternative way to create sharedareas from internal objects. This functions returns the original object (currently implemented only on CPython on top of bytearrays using `--py-sharedarea <size>` option)

6.13.5 Websockets integration API

This is currently supported only in the psgi/perl plugin:

websocket_send_from_sharedarea(id, pos) send a websocket message directly from the specified sharedarea

websocket_send_binary_from_sharedarea(id, pos) send a websocket binary message directly from the specified sharedarea

6.13.6 Advanced usage (from C)

Work in progress.

Check https://github.com/unbit/uwsgi-capture for an example of sharedarea managed from C

6.14 The uWSGI Signal Framework

Warning: Raw usage of uwsgi signals is for advanced users only. You should see *uWSGI API - Python decorators* for a more elegant abstraction.

Note: uWSGI Signals have _nothing_ in common with UNIX/Posix signals (if you are looking for those, *Managing the uWSGI server* is your page).

Over time, your uWSGI stack is growing, you add spoolers, more processes, more plugins, whatever. The more features you add the more you need all of these components to speak to each other.

Another important task for today's rich/advanced web apps is to respond to different events. An event could be a file modification, a new cluster node popping up, another one (sadly) dying, a timer having elapsed... whatever you can imagine.

Communication and event management are all managed by the same subsystem – the uWSGI signal framework.

uWSGI signals are managed with sockets, so they are *fully reliable*. When you send an uWSGI signal, you can be sure that it will be delivered.

6.14.1 The Signals table

Signals are simple *1 byte* messages that are routed by the master process to workers and spoolers.

When a worker receives a signal it searches the signals table for the corresponding handler to execute.

The signal table is shared by all workers (and protected against race conditions by a shared lock).

Every uWSGI process (mainly the master though) can write into it to set signal handlers and recipient processes.

Warning: Always pay attention to who will run the signal handler. It must have access to the handler itself. This means that if you define a new function in `worker1` and register it as a signal handler, only `worker1` can run it. The best way to register signals is defining them in the master, so (thanks to `fork()`) all workers see them.

6.14.2 Defining signal handlers

To manage the signals table the uWSGI API exposes one simple function, `uwsgi.register_signal()`.

These are two simple examples of defining signal table items, in Python and Lua.

```
import uwsgi

def hello_signal(num):
    print "i am the signal %d" % num

def hello_signal2(num):
    print "Hi, i am the signal %d" % num

# define 2 signal table items (30 and 22)
uwsgi.register_signal(30, "worker", hello_signal)
uwsgi.register_signal(22, "workers", hello_signal2)
```

```
function hello_signal(sig)
    print("i am Lua, received signal " .. sig ..)
end

# define a single signal table item (signal 1)
uwsgi.register_signal(1, "worker", hello_signal)
```

6.14.3 Raising signals

Signals may be raised using `uwsgi.signal()`. When you send a signal, it is copied into the master's queue. The master will then check the signal table and dispatch the messages.

6.14.4 External events

The most useful feature of uWSGI signals is that they can be used to announce external events.

At the time of writing the available external events are

- filesystem modifications
- timers/rb_timers
- cron

Filesystem modifications

To map a specific file/directory modification event to a signal you can use `uwsgi.add_file_monitor()`.

An example:

```
import uwsgi

def hello_file(num):
        print "/tmp has been modified !!!"

uwsgi.register_signal(17, "worker", hello_file)
uwsgi.add_file_monitor(17, "/tmp")
```

From now on, every time `/tmp` is modified, signal 17 will be raised and `hello_file` will be run by the first available worker.

Timers

Timers are another useful feature in web programming – for instance to clear sessions and shopping carts and what-have-you.

Timers are implemented using kernel facilities (most notably kqueue on BSD systems and timerfd() on modern Linux kernels). uWSGI also contains support for rb_timers, timers implemented in user space using red-black trees.

To register a timer, use `uwsgi.add_timer()`. To register an rb_timer, use `uwsgi.add_rb_timer()`.

```
import uwsgi

def hello_timer(num):
        print "2 seconds elapsed, signal %d raised" % num

def oneshot_timer(num):
        print "40 seconds elapsed, signal %d raised. You will never see me again." % num


uwsgi.register_signal(26, "worker", hello_timer)
uwsgi.register_signal(30, "", oneshot_timer)

uwsgi.add_timer(26, 2) # never-ending timer every 2 seconds
uwsgi.add_timer(30, 40, 1) # one shot timer after 40 seconds
```

Signal 26 will be raised every 2 seconds and handled by the first available worker. Signal 30 will be raised after 40 seconds and executed only once.

6.14.5 signal_wait and signal_received

Unregistered signals (those without an handler associated) will be routed to the first available worker to use the `uwsgi.signal_wait()` function.

```
uwsgi.signal_wait()
signum = uwsgi.signal_received()
```

You can combine external events (file monitors, timers...) with this technique to implement event-based apps. A good example is a chat server where every core waits for text sent by users.

You can also wait for specific (even registered) signals by passing a signal number to `signal_wait`.

6.14.6 Todo

- Signal table entry cannot be removed (this will be fixed soon)

- Iterations works only with rb_timers
- uwsgi.signal_wait() does not work in async mode (will be fixed)
- Cluster nodes popup/die signals are still not implemented.
- Bonjour/avahi/MDNS event will be implemented in 0.9.9
- PostgreSQL notifications will be implemented in 0.9.9
- Add iterations to file monitoring (to allow one-shot event as timers)

6.15 The uWSGI Spooler

Updated to uWSGI 2.0.1

Supported on: Perl, Python, Ruby

The Spooler is a queue manager built into uWSGI that works like a printing/mail system.

You can enqueue massive sending of emails, image processing, video encoding, etc. and let the spooler do the hard work in background while your users get their requests served by normal workers.

A spooler works by defining a directory in which "spool files" will be written, every time the spooler find a file in its directory it will parse it and will run a specific function.

You can have multiple spoolers mapped to different directories and even multiple spoolers mapped to the same one.

The `--spooler <directory>` option allows you to generate a spooler process, while the `--spooler-processes <n>` allows you to set how many processes to spawn for every spooler.

The spooler is able to manage uWSGI signals too, so you can use it as a target for your handlers.

This configuration will generate a spooler for your instance (myspool directory must exists)

```
[uwsgi]
spooler = myspool
...
```

while this one will create two spoolers:

```
[uwsgi]
spooler = myspool
spooler = myspool2
...
```

having multiple spoolers allows you to prioritize tasks (and eventually parallelize them)

6.15.1 Spool files

Spool files are serialized hash/dictionary of strings. The spooler will parse them and pass the resulting hash/dictionary to the spooler function (see below).

The serialization format is the same used for the 'uwsgi' protocol, so you are limited to 64k (even if there is a trick for passing bigger values, see the 'body' magic key below). The modifier1 for spooler packets is the 17, so a {'hello' => 'world'} hash will be encoded as:

header	key1	value1
17|14|0|0	|5|0|h|e|l|l|o	|5|0|w|o|r|l|d

A locking system allows you to safely manually remove spool files if something goes wrong, or to move them between spoolers directory.

Spool dirs over NFS are allowed, but if you do not have proper NFS locking in place, avoid mapping the same spooler NFS directory to spooler on different machines.

6.15.2 Setting the spooler function/callable

To have a fully operational spooler you need to define a "spooler function/callable".

Independently by the the number of configured spoolers, the same function will be executed. It is up to the developer to instruct it to recognize tasks.

This function must returns an integer value:

-2 (SPOOL_OK) the task has been completed, the spool file will be removed

-1 (SPOOL_RETRY) something is temporarely wrong, the task will be retried at the next spooler iteration

0 (SPOOL_IGNORE) ignore this task, if multiple languages are loaded in the instance all of them will fight for managing the task. This return values allows you to skip a task in specific languages.

any other value will be mapped as -1 (retry)

Each language plugin has its way to define the spooler function:

Perl:

```
uwsgi::spooler(
    sub {
        my ($env) = @_;
        print $env->{foobar};
        return uwsgi::SPOOL_OK;
    }
);
```

Python:

```
import uwsgi

def my_spooler(env):
    print env['foobar']
    return uwsgi.SPOOL_OK

uwsgi.spooler = my_spooler
```

Ruby:

```
module UWSGI
     module_function
     def spooler(env)
             puts env.inspect
             return UWSGI::SPOOL_OK
     end
end
```

Spooler function must be defined in the master process, so if you are in lazy-apps mode, be sure to place it in a file that is parsed early in the server setup. (in python you can use –shared-import, in ruby –shared-require, in perl –perl-exec).

Some language plugin could have support for importing code directly in the spooler. Currently only python supports it with the `--spooler-import` option.

6.15.3 Enqueueing requests to a spooler

The 'spool' api function allows you to enqueue a hash/dictionary into the spooler:

```
# python
import uwsgi
uwsgi.spool({'foo': 'bar', 'name': 'Kratos', 'surname': 'the same of Zeus'})
# or
uwsgi.spool(foo='bar', name='Kratos', surname='the same of Zeus')
# for python3 use bytes instead of strings !!!

# perl
uwsgi::spool({foo => 'bar', name => 'Kratos', surname => 'the same of Zeus'})

# ruby
UWSGI.spool(foo => 'bar', name => 'Kratos', surname => 'the same of Zeus')
```

Some keys have special meaning:

'spooler' => specify the ABSOLUTE path of the spooler that has to manage this task

'at' => unix time at which the task must be executed (read: the task will not be run until the 'at' time is passed)

'priority' => this will be the subdirectory in the spooler directory in which the task will be placed, you can use that trick to give a good-enough prioritization to tasks (for better approach use multiple spoolers)

'body' => use this key for objects bigger than 64k, the blob will be appended to the serialzed uwsgi packet and passed back to the spooler function as the 'body' argument

IMPORTANT:

spool arguments must be strings (or bytes for python3), the api function will try to cast non-string values to strings/bytes, but do not rely on it !!!

6.15.4 External spoolers

You could want to implement a centralized spooler for your server.

A single instance will manage all of the tasks enqueued by multiple uWSGI servers.

To accomplish this setup each uWSGI instance has to know which spooler directories are valid (consider it a form of security)

To add an external spooler directory use the `--spooler-external <directory>` option.

The spooler locking subsystem will avoid mess

6.15.5 Networked spoolers

You can even enqueue tasks over the network (be sure the 'spooler' plugin is loaded in your instance, but generally it is builtin by default)

As we have already seen, spooler packets have modifier1 17, you can directly send those packets to a uwsgi socket of an instance with a spooler enabled.

We will use the perl Net::uwsgi module (exposing a handy uwsgi_spool function) in this example (feel free to use whatever you want):

```
use Net::uwsgi;
uwsgi_spool('localhost:3031', {'test'=>'test001','argh'=>'boh','foo'=>'bar'});
uwsgi_spool('/path/to/socket', {'test'=>'test001','argh'=>'boh','foo'=>'bar'});

[uwsgi]
socket = /var/run/uwsgi-spooler.sock
socket = localhost:3031
spooler = /path/for/files
spooler-processes=1
...
```

(thanks brianhorakh for the example)

6.15.6 Priorities

We have already seen that you can use the 'priority' key to give order in spooler parsing.

While having multiple spoolers would be an extremely better approach, on system with few resources 'priorities' are a good trick

They works only if you enable the `--spooler-ordered` option. This option allows the spooler to scan directories entry in alphabetical order.

If during the scan a directory with a 'number' name is found, the scan is suspended and the content of this subdirectory will be explored for tasks.

```
/spool
/spool/ztask
/spool/xtask
/spool/1/task1
/spool/1/task0
/spool/2/foo
```

with this layout the order in which files will be parsed is:

```
/spool/1/task0
/spool/1/task1
/spool/2/foo
/spool/xtask
/spool/ztask
```

remember, priorities only works for subdirectory named as 'numbers' and you need the `--spooler-ordered` option.

The uWSGI spooler gives special names to tasks so the ordering of enqueuing is always respected.

6.15.7 Options

`spooler=directory` run a spooler on the specified directory

`spooler-external=directory` map spoolers requests to a spooler directory managed by an external instance

`spooler-ordered` try to order the execution of spooler tasks (uses scandir instead of readdir)

`spooler-chdir=directory` call chdir() to specified directory before each spooler task

`spooler-processes=##` set the number of processes for spoolers

`spooler-quiet` do not be verbose with spooler tasks

`spooler-max-tasks=##` set the maximum number of tasks to run before recycling a spooler (to help alleviate memory leaks)

`spooler-harakiri=##` set harakiri timeout for spooler tasks, see [harakiri] for more information.

`spooler-frequency=##` set the spooler frequency

6.15.8 Tips and tricks

You can re-enqueue a spooler request by returning `uwsgi.SPOOL_RETRY` in your callable:

```
def call_me_again_and_again(env):
    return uwsgi.SPOOL_RETRY
```

You can set the spooler poll frequency using the `--spooler-frequency <secs>` option (default 30 seconds).

You could use the *The uWSGI caching framework* or *SharedArea – share memory pages between uWSGI components* to exchange memory structures between spoolers and workers.

Python (uwsgidecorators.py) and Ruby (uwsgidsl.rb) exposes higher-level facilities to manage the spooler, try to use them instead of the low-level approach described here.

When using a spooler as a target for a uWSGI signal handler you can specify which one to route signal to using its ABSOLUTE directory name.

6.16 uWSGI Subscription Server

Some components of the uWSGI stack require a key-value mapping system.

For example the *The uWSGI FastRouter* needs to know which server to contact for a specific request.

In big networks with a lot of nodes manually managing this configuration could be a real hell. uWSGI implements a subscription system where the node itself announces its presence to Subscription Servers, which will in turn populate their internal dictionaries.

```
uwsgi --fastrouter :1717 --fastrouter-subscription-server 192.168.0.100:2626
```

This will run an uWSGI fastrouter on port 1717 and create an empty dictionary where the hostname is the key and the uwsgi address is the value.

To populate this dictionary you can contact 192.168.0.100:2626, the address of the subscription server.

For every key multiple addresses can exist, enabling round robin load balancing.

A node can announce its presence to a Subscription Server using the `subscribe-to` or `subscribe2` options.

```
uwsgi -s 192.168.0.10:3031 -w myapp -M --subscribe-to 192.168.0.100:2626:uwsgi.it
```

The FastRouter will map every request for uwsgi.it to 192.168.0.10:3031.

To now add a second node for uwsgi.it simply run it and subscribe:

```
uwsgi -s 192.168.0.11:3031 -w myapp -M --subscribe-to 192.168.0.100:2626:uwsgi.it
```

Dead nodes are automatically removed from the pool.

The syntax for `subscribe2` is similar but it allows far more control since it allows to specify additional options like the address to which all requests should be forwarded. Its value syntax is a string with "key=value" pairs, each separated by a comma.

```
uwsgi -s 192.168.0.10:3031 -w myapp -M --subscribe2 server=192.168.0.100:2626,key=uwsgi.it,addr=192.1
```

Possible keys are:

- `server` - address (ip:port) of the subscription server we want to connect to
- `key` - key used for mapping, hostname (FastRouter or HttpRouter) or ip:port (RawRouter)
- `socket` - TODO
- `addr` - address to which all requests should be forwared for this subscription
- `weight` - node weight for load balancing
- `modifier1` - modifier1 value for our app
- `modifier2` - modifier2 value for our app
- `sign` - for secure subscription (see below)
- `check` - if the specified file exists stop sending subscriptions

The subscription system is currently available for cluster joining (when multicast/broadcast is not available), the Fastrouter and HTTP.

That said, you can create an evented/fast_as_hell HTTP load balancer in no time.

```
uwsgi --http :80 --http-subscription-server 192.168.0.100:2626
```

Now simply subscribe your nodes to the HTTP subscription server.

6.16.1 Securing the Subscription System

The subscription system is meant for "trusted" networks. All of the nodes in your network can potentially make a total mess with it.

If you are building an infrastructure for untrusted users or you simply need more control over who can subscribe to a Subscription Server you can use openssl rsa public/private key pairs for "signing" you subscription requests.

```
# First, create the private key for the subscriber. DO NOT SET A PASSPHRASE FOR THIS KEY.
openssl genrsa -out private.pem
# Generate the public key for the subscription server:
openssl rsa -pubout -out test.uwsgi.it_8000.pem -in private.pem
```

The keys must be named after the domain/key we are subscribing to serve, plus the .pem extension.

Note: If you're subscribing to a pool for an application listening on a specified port you need to use the `domain_port.pem` scheme for your key files. Generally all of the DNS-allowed chars are supported, all of the others are mapped to an underscore.

An example of an RSA protected server looks like this:

```
[uwsgi]
master = 1
http = :8000
http-subscription-server = 127.0.0.1:2626
subscriptions-sign-check = SHA1:/etc/uwsgi/keys
```

The last line tells uWSGI that public key files will be stored in /etc/uwsgi/keys.

At each subscription request the server will check for the availability of the public key file and use it, if available, to verify the signature of the packet. Packets that do not correctly verify are rejected.

On the client side you need to pass your private key along with other `subscribe-to` options. Here's an example:

```
[uwsgi]
socket = 127.0.0.1:8080
subscribe-to = 127.0.0.1:2626:test.uwsgi.it:8000,5,SHA1:/home/foobar/private.pem
psgi = test.psgi
```

Let's analyze the `subscribe-to` usage:

- `127.0.0.1:2626` is the subscription server we want to subscribe to.
- `test.uwsgi.it:8000` is the subscription key.
- `5` is the modifier1 value for our psgi app
- `SHA1:/home/private/test.uwsgi.it_8000.pem` is the <digest>:<rsa> couple for authenticating to the server (the <rsa> field is the private key path).

Note: Please make sure you're using the same digest method (SHA1 in the examples above) both on the server and on the client.

To avoid replay attacks, each subscription packet has an increasing number (normally the unix time) avoiding the allowance of duplicated packets. Even if an attacker manages to sniff a subscription packet it will be unusable as it is already processed previously. Obviously if someone manages to steal your private key he will be able to build forged packets.

Using SSH keys

They are generally loved by developers (well, more than classic pem files).

Both –subscribe-to and –subscribe2 (see below) support ssh private keys, while for the server part you have the encode the public key in pkcs8:

```
ssh-keygen -f chiavessh001.pub -e -m pkcs8
```

6.16.2 –subscribe2

This is the keyval version of –subscribe-to. It supports more tricks and a (generally) more readable syntax:

```
uwsgi -s 127.*:0 --subscribe2 server=127.0.0.1:7171,key=ubuntu64.local:9090,sign=SHA1:chiavessh001
```

Supported fields are:

`server` the address of the subscription server

`key` the key to subscribe (generally the domain name)

`addr` the address to subscribe (the value of the item)

`socket` the socket number (zero-based), this is like 'addr' by take the uWSGI internal socket number

`weight` the load balancing value

`modifier1` and `modifier2`

`sign` <algo>:<file> the signature for the secured system

`check` it takes a file as argument. If it exists the packet is sent, otherwise it is skipped

6.17 Serving static files with uWSGI (updated to 1.9)

Unfortunately you cannot live without serving static files via some protocol (HTTP, SPDY or something else).

Fortunately uWSGI has a wide series of options and micro-optimizations for serving static files.

Generally your webserver of choice (Nginx, Mongrel2, etc. will serve static files efficiently and quickly and will simply forward dynamic requests to uWSGI backend nodes.

The uWSGI project has ISPs and PaaS (that is, the hosting market) as the main target, where generally you would want to avoid generating disk I/O on a central server and have each user-dedicated area handle (and account for) that itself. More importantly still, you want to allow customers to customize the way they serve static assets without bothering your system administrator(s).

6.17.1 Mode 1: Check for a static resource before passing the request to your app

This a fairly common way of managing static files in web apps. Frameworks like Ruby on Rails and many PHP apps have used this method for ages.

Suppose your static assets are under `/customers/foobar/app001/public`. You want to check each request has a corresponding file in that directory before passing the request to your dynamic app. The `--check-static` option is for you:

```
--check-static /customers/foobar/app001/public
```

If uWSGI receives a request for `/foo.png` will first check for the existence of `/customers/foobar/app001/public/foo.png` and if it is not found it will invoke your app.

You can specify `--check-static` multiple times to specify multiple possible root paths.

```
--check-static /customers/foobar/app001/public --check-static /customers/foobar/app001/static
```

uWSGI will first check for `/customers/foobar/app001/public/foo.png`; if it does not find it it will try `/customers/foobar/app001/static/foo.png` before finally delegating the request to your app.

6.17.2 Mode 2: trust frontend's DOCUMENT_ROOT

If your frontend (a webserver, a uWSGI corerouters...) set the `DOCUMENT_ROOT` value, you can instruct uWSGI to trust it as a valid directory for checking for static files with the `--check-static-docroot` option.

6.17.3 Mode 3: using static file mount points

A more general approach is "mapping" specific request prefixes to physical directories on your file system.

The `--static-map mountpoint=path` is the option for you.

```
--static-map /images=/var/www/img
```

If you get a request for `/images/logo.png` and `/var/www/img/logo.png` exists, it will be served. Otherwise your app will manage the request.

You can specify multiple `--static-map` options, even for the same mountpoint.

```
--static-map /images=/var/www/img --static-map /images=/var/www/img2 --static-map /images=/var/www/ir
```

The file will be searched in each directory until it's found, or if it's not, the request will be managed by your app.

In some specific cases you may want to build the internal path in a different way, retaining the original path portion of the request. The `--static-map2` option will do this.

```
--static-map2 /images=/var/www/img
```

A request for `/images/logo.png` will be looked for as `/var/www/img/images/logo.png`.

You can `map` (or `map2`) both directories and files.

```
--static-map /images/logo.gif=/tmp/oldlogo.gif
# (psst: put favicons here)
```

6.17.4 Mode 4: using advanced internal routing

When mappings are not enough, advanced internal routing (available from 1.9) will be your last resort.

Thanks to the power of regular expressions you will be able to build very complex mappings.

```
[uwsgi]
route = /static/(.*)\.png static:/var/www/images/pngs/$1/highres.png
route = *\.jpg static:/var/www/always_the_same_photo.jpg
```

6.17.5 Setting the index page

By default, requests for a "directory" (like / or /foo) are bypassed (if advanced internal routing is not in place).

If you want to map specific files to a "directory" request (like the venerable index.html) just use the `--static-index` option.

```
--static-index index.html --static-index index.htm --static-index home.html
```

As with the other options, the first one matching will stop the chain.

6.17.6 MIME types

Your HTTP/SPDY/whateveryouwant responses for static files should always return the correct mime type for the specific file to let user agents handle them correctly.

By default uWSGI builds its list of MIME types from the `/etc/mime.types` file. You can load additional files with the `--mime-file` option.

```
--mime-file /etc/alternatives.types --mime-file /etc/apache2/mime.types
```

All of the files will be combined into a single auto-optimizing linked list.

6.17.7 Skipping specific extensions

Some platforms/languages, most-notably CGI based ones, like PHP are deployed in a very simple manner.

You simply drop them in the document root and they are executed whenever you call them.

This approach, when combined with static file serving, requires a bit of attention for avoiding your CGI/PHP/whatever to be served like static files.

The `--static-skip-ext` will do the trick.

A very common pattern on CGI and PHP deployment is this:

```
--static-skip-ext .php --static-skip-ext .cgi --static-skip-ext .php4
```

6.17.8 Setting the Expires headers

When serving static files, abusing client browser caching is the path to wisdom. By default uWSGI will add a `Last-Modified` header to all static responses, and will honor the `If-Modified-Since` request header.

This might be not enough for high traffic sites. You can add automatic `Expires` headers using one of the following options:

- `--static-expires-type` will set the Expires header to the specified number of seconds for the specified MIME type.
- `--static-expires-type-mtime` is similar, but based on file modification time, not the current time.
- `--static-expires` (and `-mtime`) will set Expires header for all of the filenames (after finishing mapping to the filesystem) matching the specified regexp.
- `--static-expires-uri` (and `-mtime`) match regexps against `REQUEST_URI`
- `--static-expires-path-info` (and `-mtime`) match regexps against `PATH_INFO`

```
# Expire an hour from now
--static-expires-type text/html=3600
# Expire an hour from the file's modification time
--static-expires-type-mtime text/html=3600
# Same as static-expires-type, but based on a regexp:
--static-expires /var/www/static/foo*\.jpg 3600
```

6.17.9 Transfer modes

If you have developed an asynchronous/nonblocking application, serving static files directly from uWSGI is not a big problem.

All of the transfers are managed in the async way, so your app will not block during them.

In multi-process/multi-threaded modes, your processes (or threads) will be blocked during the whole transfer of the file.

For smaller files this is not a problem, but for the bigger one it's a great idea to offload their transfer to something else.

You have various ways to do this:

X-Sendfile

If your web server supports the X-Sendfile header and has access to the file you want to send (for example it is on the same machine of your application or can access it via NFS) you can avoid the transfer of the file from your app with the `--file-serve-mode x-sendfile` option.

With this, uWSGI will only generate response headers and the web server will be delegated to transferring the physical file.

X-Accel-Redirect

This is currently (January 2013) supported only on Nginx. Works in the same way as X-Sendfile, the only difference is in the option argument.

```
--file-serve-mode x-accel-redirect
```

Offloading

This is the best approach if your frontend server has no access to the static files. It uses the *The uWSGI offloading subsystem* to delegate the file transfer to a pool of non-blocking threads.

Each one of these threads can manage thousands of file transfers concurrently.

To enable file transfer offloading just use the option `--offload-threads` specifying the number of threads to spawn (try to set it to the number of CPU cores to take advantage of SMP).

6.17.10 GZIP (uWSGI 1.9)

uWSGI 1.9 can check for a `*.gz` variant of a static file.

Many users/sysadmins underestimate the CPU impact of on-the-fly Gzip encoding.

Compressing files every time (unless your webservers is caching them in some way) will use CPU and you will not be able to use advanced (zero-copy) techniques like `sendfile()`. For a very loaded site (or network) this could be a problem (especially when gzip encoding is a need for a better, more responsive user experience).

Although uWSGI is able to compress contents on the fly (this is used in the HTTP/HTTPS/SPDY router for example), the best approach for serving gzipped static files is generating them "manually" (but please use a script, not an intern to do this), and let uWSGI choose if it is best to serve the uncompressed or the compressed one every time.

In this way serving gzip content will be no different from serving standard static files (sendfile, offloading...)

To trigger this behavior you have various options:

- `static-gzip <regexp>` checks for .gz variant for all of the requested files matching the specified regexp (the regexp is applied to the full filesystem path of the file)
- `static-gzip-dir <dir>`/`static-gzip-prefix <prefix>` checks for .gz variant for all of the files under the specified directory
- `static-gzip-ext <ext>`/`static-gzip-suffix <suffix>` check for .gz variant for all of the files with the specified extension/suffix
- `static-gzip-all` check for .gz variant for all requested static files

So basically if you have `/var/www/uwsgi.c` and `/var/www/uwsgi.c.gz`, clients accepting gzip as their Content-Encoding will be transparently served the gzipped version instead.

6.17.11 Security

Every static mapping is fully translated to the "real" path (so symbolic links are translated too).

If the resulting path is not under the one specified in the option, a security error will be triggered and the request refused.

If you trust your UNIX skills and know what you are doing, you can add a list of "safe" paths. If a translated path is not under a configured directory but it is under a safe one, it will be served nevertheless.

Example:

```
--static-map /foo=/var/www/
```

`/var/www/test.png` is a symlink to `/tmp/foo.png`

After the translation of `/foo/test.png`, uWSGI will raise a security error as `/tmp/foo.png` is not under `/var/www/`.

Using

```
--static-map /foo=/var/www/ --static-safe /tmp
```

will bypass that limit.

You can specify multiple `--static-safe` options.

6.17.12 Caching paths mappings/resolutions

One of the bottlenecks in static file serving is the constant massive amount of `stat()` syscalls.

You can use the uWSGI caching subsystem to store mappings from URI to filesystem paths.

```
--static-cache-paths 30
```

will cache each static file translation for 30 seconds in the uWSGI cache.

From uWSGI 1.9 an updated caching subsystem has been added, allowing you to create multiple caches. If you want to store translations in a specific cache you can use `--static-cache-paths-name <cachename>`.

6.17.13 Bonus trick: storing static files in the cache

You can directly store a static file in the uWSGI cache during startup using the option `--load-file-in-cache <filename>` (you can specify it multiple times). The content of the file will be stored under the key <filename>.

So please pay attention – `load-file-in-cache ./foo.png` will store the item as `./foo.png`, not its full path.

6.17.14 Notes

- The static file serving subsystem automatically honours the If-Modified-Since HTTP request header

6.18 SNI - Server Name Identification (virtual hosting for SSL nodes)

uWSGI 1.9 (codenamed "ssl as p0rn") added support for SNI (Server Name Identification) throughout the whole SSL subsystem. The HTTPS router, the SPDY router and the SSL router can all use it transparently.

SNI is an extension to the SSL standard which allows a client to specify a "name" for the resource it wants. That name is generally the requested hostname, so you can implement virtual hosting-like behavior like you do using the HTTP `Host:` header without requiring extra IP addresses etc.

In uWSGI an SNI object is composed of a name and a value. The name is the servername/hostname while the value is the "SSL context" (you can think of it as the sum of certificates, key and ciphers for a particular domain).

6.18.1 Adding SNI objects

To add an SNI object just use the `--sni` option:

```
--sni <name> crt,key[,ciphers,client_ca]
```

For example:

```
--sni "unbit.com unbit.crt,unbit.key"
```

or for client-based SSL authentication and OpenSSL HIGH cipher levels

```
--sni "secure.unbit.com unbit.crt,unbit.key,HIGH,unbit.ca"
```

6.18.2 Adding complex SNI objects

Sometimes you need more complex keys for your SNI objects (like when using wildcard certificates)

If you have built uWSGI with PCRE/regexp support (as you should) you can use the `--sni-regexp` option.

```
--sni-regexp "*.unbit.com unbit.crt,unbit.key,HIGH,unbit.ca"
```

6.18.3 Massive SNI hosting

One of uWSGI's main purposes is massive hosting, so SSL without support for that would be pretty annoying.

If you have dozens (or hundreds, for that matter) of certificates mapped to the same IP address you can simply put them in a directory (following a simple convention we'll elaborate in a bit) and let uWSGI scan it whenever it needs to find a context for a domain.

To add a directory just use

```
--sni-dir <path>
```

like

```
--sni-dir /etc/customers/certificates
```

Now, if you have `unbit.com` and `example.com` certificates (.crt) and keys (.key) just drop them in there following these naming rules:

- `/etc/customers/certificates/unbit.com.crt`
- `/etc/customers/certificates/unbit.com.key`
- `/etc/customers/certificates/unbit.com.ca`
- `/etc/customers/certificates/example.com.crt`
- `/etc/customers/certificates/example.com.key`

As you can see, `example.com` has no .ca file, so client authentication will be disabled for it.

If you want to force a default cipher set to the SNI contexts, use

```
--sni-dir-ciphers HIGH
```

(or whatever other value you need)

Note: Unloading SNI objects is not supported. Once they are loaded into memory they will be held onto until reload.

6.18.4 Subscription system and SNI

uWSGI 2.0 added support for SNI in the subscription system.

The https/spdy router and the sslrouter can dinamically load certificates and keys from the paths specified in a subscription packet:

```
uwsgi --subscribe2 key=mydomain.it,socket=0,sni_key=/foo/bar.key,sni_cert=/foo/bar.crt
```

the router will create a new SSL context based on the specified files (be sure the router can reach them) and will destroy it when the last node disconnect.

This is useful for massive hosting where customers have their certificates in the home and you want them the change/update those files without bothering you.

We understand that directly encapsulating keys and cert in the subscription packets will be much more useful, but network transfer of keys is something really foolish from a security point of view. We are investigating if combining it with the secured subscription system (where each packet is encrypted) could be a solution.

6.19 The GeoIP plugin

The `geoip` plugin adds new routing vars to your internal routing subsystem. GeoIP's vars are prefixed with the "geoip" tag. To build the geoip plugin you need the official GeoIP C library and its headers. The supported databases are the country and city one, and they are completely loaded on memory at startup.

The country database give access to the following variables:

- `${geoip[country_code]}`
- `${geoip[country_code3]}`
- `${geoip[country_name]}`

while the city one offers a lot more at the cost of increased memory usage for storing the database

- ``${geoip[continent]}
- ``${geoip[country_code]}
- ``${geoip[country_code3]}
- ``${geoip[country_name]}
- ``${geoip[region]}
- ``${geoip[region_name]}
- ``${geoip[city]}
- ``${geoip[postal_code]}
- `${geoip[latitude]}` (``${geoip[lat]}`)
- `${geoip[longitude]}` (``${geoip[lon]}`)
- ``${geoip[dma]}
- ``${geoip[area]}

6.19.1 Enabling geoip lookup

To enable the GeoIP lookup system you need to load at least one database. After having loaded the geoip plugin you will get 2 new options:

- `--geoip-country` specifies a country database
- `--geoip-city` specifies a city database

If you do not specify at least one of them, the system will always return empty strings.

6.19.2 An example

```
[uwsgi]
plugin = geoip
http-socket = :9090
; load the geoip city database
geoip-city = GeoLiteCity.dat
module = werkzeug.testapp:test_app
; first some debug info (addvar will ad WSGI variables you will see in the werkzeug testapp)
route-run = log:${geoip[country_name]}/${geoip[country_code3]}
route-run = addvar:COUNTRY=${geoip[country_name]}
route-run = log:${geoip[city]}/${geoip[region]}/${geoip[continent]}
route-run = addvar:COORDS=${geoip[lon]}/${geoip[lat]}
route-run = log:${geoip[region_name]}
route-run = log:${geoip[dma]}/${geoip[area]}

; then something more useful
; block access to all of the italians (hey i am italian do not start blasting me...)
route-if = equal:${geoip[country_name]};Italy break:403 Italians cannot see this site :P
; try to serve a specific page translation
route = ^/foo/bar/test.html static:/var/www/${geoip[country_code]}/test.html
```

6.19.3 Memory usage

The country database is tiny so you will generally have no problem in using it. Instead, the city database can be huge (from 20MB to more than 40MB). If you have lot of instances using the GeoIP city database and you are on a recent Linux system, consider using *Using Linux KSM in uWSGI* to reduce memory usage. All of the memory used by the GeoIP database can be shared by all instances with it.

6.20 uWSGI Transformations

Starting from uWSGI 1.9.7, a "transformations" API has been added to *uWSGI internal routing*.

A transformation is like a filter applied to the response generated by your application.

Transformations can be chained (the output of a transformation will be the input of the following one) and can completely overwrite response headers.

The most common example of transformation is gzip encoding. The output of your application is passed to a function compressing it with gzip and setting the Content-Encoding header. This feature rely on 2 external packages: libpcre3-dev, libz-dev on Ubuntu.

```
[uwsgi]
plugin = python,transformation_gzip
http-socket = :9090
; load the werkzeug test app
module = werkzeug.testapp:test_app
; if the client supports gzip encoding goto to the gzipper
route-if = contains:${HTTP_ACCEPT_ENCODING};gzip goto:mygzipper
route-run = last:

route-label = mygzipper
; pass the response to the gzip transformation
route = ^/$ gzip:
```

The `cachestore` routing instruction is a transformation too, so you can cache various states of the response.

```
[uwsgi]
plugin = python,transformation_gzip
http-socket = :9090
; load the werkezeug test app
module = werkzeug.testapp:test_app
; create a cache of 100 items
cache = 100
; if the client support gzip encoding goto to the gzipper
route-if = contains:${HTTP_ACCEPT_ENCODING};gzip goto:mygzipper
route = ^/$ cache:key=werkzeug_homepage
route = ^/$ cachestore:key=werkzeug_homepage
route-run = last:

route-label = mygzipper
route = ^/$ cache:key=werkzeug_homepage.gz
; first cache the 'clean' response (for client not supporting gzip)
route = ^/$ cachestore:key=werkzeug_homepage
; then pass the response to the gzip transformation
route = ^/$ gzip:
; and cache it again in another item (gzipped)
route = ^/$ cachestore:key=werkzeug_homepage.gz
```

Another common transformation is applying stylesheets to XML files. (see *The XSLT plugin*)

The `toxslt` transformation is exposed by the `xslt` plugin:

```
uwsgi --plugin xslt --http-socket :9090 -w mycd --route-run "toxslt:stylesheet=t/xslt/cd.xml.xslt,pa
```

The `mycd` module here is a simple XML generator. Its output is then passed to the XSLT transformation.

6.20.1 Streaming vs. buffering

Each transformation announces itself as a "streaming" one or a "buffering" one.

Streaming ones are transformations that can be applied to response chunks (parts). An example of a streaming transformation is gzip (you do not need the whole body to begin compressing it). Buffering transformations are those requiring the full body before applying something to it. XSLT is an example of buffering transformation. Another example of buffering transformations are those used for storing response in some kind of cache.

If your whole pipeline is composed by only "streaming" transformations, your client will receive the output chunk by chunk. On the other hand a single buffering transformation will make the whole pipeline buffered, so your client will get the output only at the end.

An often using streaming functionality is gzip + chunked:

```
[uwsgi]
plugins = transformation_gzip,transformation_chunked
route-run = gzip:
route-run = chunked:
...
```

The whole transformation pipeline is composed by streaming plugins, so you will get each HTTP chunk in realtime.

6.20.2 Flushing magic

The "flush" transformation is a special one. It allows you to send the current contents of the transformation buffer to the client (without clearing the buffer).

You can use it for implementing streaming mode when buffering will be applied. A common example is having streaming + caching:

```
[uwsgi]
plugins = transformation_toupper,transform_tofile
; convert each char to uppercase
route-run = toupper:
; after each chunk converted to upper case, flush to the client
route-run = flush:
; buffer the whole response in memory for finally storing it in a file
route-run = tofile:filename=/tmp/mycache
...
```

You can call flush multiple times and in various parts of the chain. Experiment a bit with it...

6.20.3 Available transformations (last update 20130504)

- `gzip`, exposed by the `transformation_gzip` plugin (encode the response buffer to gzip)
- `toupper`, exposed by the `transformation_toupper` plugin (example plugin transforming each character in uppercase)
- `tofile`, exposed by the `transformation_tofile` plugin (used for caching to response buffer to a static file)
- `toxslt`, exposed by the `xslt` plugin (apply xslt stylesheet to an XML response buffer)
- `cachestore`, exposed by the `router_cache` plugin (cache the response buffer in the uWSGI cache)
- `chunked`, encode the output in HTTP chunked
- `flush`, flush the current buffer to the client
- `memcachedstore`, store the response buffer in a memcached object
- `redisstore`, store the response buffer in a redis object
- `template`, apply routing translations to each chunk

6.20.4 Working on

- `rpc`, allows applying rpc functions to a response buffer (limit 64k size)
- `lua`, apply a lua function to a response buffer (no limit in size)

6.21 WebSocket supports

In uWSGI 1.9, a high performance websocket (RFC 6455) implementation has been added.

Although many different solutions exist for WebSockets, most of them rely on a higher-level language implementation, that rarely is good enough for topics like gaming or streaming.

The uWSGI websockets implementation is compiled in by default

Websocket support is sponsored by 20Tab S.r.l. http://20tab.com/

They released a full game (a bomberman clone based on uWSGI websockets api): https://github.com/20tab/Bombertab

6.21.1 An echo server

This is how a uWSGI websockets application looks like:

```
def application(env, start_response):
    # complete the handshake
    uwsgi.websocket_handshake(env['HTTP_SEC_WEBSOCKET_KEY'], env.get('HTTP_ORIGIN', ''))
    while True:
        msg = uwsgi.websocket_recv()
        uwsgi.websocket_send(msg)
```

You do not need to worry about keeping the connection alive or reject dead peers. The `uwsgi.websocket_recv()` function will do all of the dirty work for you in background.

6.21.2 Handshaking

Handshaking is the first phase of a websocket connection.

To send a full handshake response you can use the `uwsgi.websocket_handshake([key,origin, proto])` function. Without a correct handshake the connection will never complete.

In 1.9 serie, the key parameter is required, while in 2.0 you can call websocket_handshake without arguments (response will be build automatically from request's data)

6.21.3 Sending

Sending data to the browser is really easy. `uwsgi.websocket_send(msg)` – nothing more.

6.21.4 Receiving

This is the real core of the whole implementation.

This function actually lies about its real purpose. It does return a websocket message, but it really also holds the connection opened (using the ping/pong subsystem) and monitors the stream's status.

`msg = uwsgi.websocket_recv()`

The function can receive messages from a named channel (see below) and automatically forward them to your websocket connection.

It will always return only websocket messages sent from the browser – any other communication happens in the background.

There is a non-blocking variant too:

```
msg = uwsgi.websocket_recv_nb()
```

check: https://github.com/unbit/uwsgi/blob/master/tests/websockets_chat_async.py

6.21.5 PING/PONG

To keep a websocket connection opened, you should constantly send ping (or pong, see later) to the browser and expect a response from it. If the response from the browser/client does not arrive in a timely fashion the connection is closed (`uwsgi.websocket_recv()` will raise an exception). In addition to ping, the `uwsgi.websocket_recv()` function send the so called 'gratuitous pong'. They are used to inform the client of server availability.

All of these tasks happen in background. YOU DO NOT NEED TO MANAGE THEM!

6.21.6 Available proxies

Unfortunately not all of the HTTP webserver/proxies work flawlessly with websockets.

- The uWSGI HTTP/HTTPS/SPDY router supports them without problems. Just remember to add the `--http-websockets` option.

  ```
  uwsgi --http :8080 --http-websockets --wsgi-file myapp.py
  ```

or

```
uwsgi --http :8080 --http-raw-body --wsgi-file myapp.py
```

it is a bit more rawer but supports things like chunked input

- Haproxy works fine.
- nginx >= 1.4 works fine and without additional configuration

6.21.7 Languages support

- Python https://github.com/unbit/uwsgi/blob/master/tests/websockets_echo.py
- Perl https://github.com/unbit/uwsgi/blob/master/tests/websockets_echo.pl
- PyPy https://github.com/unbit/uwsgi/blob/master/tests/websockets_chat_async.py
- Ruby https://github.com/unbit/uwsgi/blob/master/tests/websockets_echo.ru
- Lua https://github.com/unbit/uwsgi/blob/master/tests/websockets_echo.lua

6.21.8 Supported Concurrency models

- Multiprocess
- Multithreaded
- uWSGI native async api
- Coro::AnyEvent
- gevent
- Ruby fibers + uWSGI async

- Ruby threads
- greenlet + uWSGI async
- uGreen + uWSGI async
- PyPy continulets

6.21.9 wss:// (websockets over https)

The uWSGI HTTPS router works without problems with websockets. Just remember to use wss:// as the connection scheme in your client code.

6.21.10 Websockets over SPDY

n/a

6.21.11 Routing

The http proxy internal router supports websocket out of the box (assuming your front-line proxy already supports them)

```
[uwsgi]
route = ^/websocket uwsgi:127.0.0.1:3032,0,0
```

or

```
[uwsgi]
route = ^/websocket http:127.0.0.1:8080
```

6.21.12 Api

uwsgi.websocket_handshake([key, origin, proto])

uwsgi.websocket_recv()

uwsgi.websocket_send(msg)

uwsgi.websocket_send_binary(msg) (added in 1.9.21 to support binary messages)

uwsgi.websocket_recv_nb()

uwsgi.websocket_send_from_sharedarea(id, pos) (added in 1.9.21, allows sending directly from a *SharedArea – share memory pages between uWSGI components*)

uwsgi.websocket_send_binary_from_sharedarea(id, pos) (added in 1.9.21, allows sending directly from a *SharedArea – share memory pages between uWSGI components*)

6.22 The Metrics subsystem

(available from 1.9.19)

The uWSGI metrics subsystem allows you to manage “numbers” from your app.

While the caching subsystem got math capabilities during 1.9 development cycle, the metrics subsystem is optimized by design for storing numbers and applying functions over them. So compared to the caching subsystem is way faster and requires a fraction of the memory.

When enabled, the metric subsystem configures a vast amount of metrics (like requests per-core, memory usage...) but, in addition to this, you can configure your own metrics (for example you can account the number users or the hit of a particular url, as well as the memory consumption of your app or the whole server)

To enable the metrics subsystem just add `--enable-metrics` to your options, or configure a stats pusher (see below).

The metrics subsystem is totally thread-safe

By default uWSGI creates a lot of metrics (and mores are planned) so before adding your own, be sure uWSGI does not already expose the one you need.

6.22.1 Metric names and oids

Each metric must have a name (containing only numbers, letters, underscores, dashes and dots) and an optional oid.

The oid is required if you want to map a metric to *The embedded SNMP server*

6.22.2 Metric types

Before dealing with metrics you need to understand the various types represented by each metric:

COUNTER (type 0)

this is a generally-growing up number (like the number of requests)

GAUGE (type 1)

this is a number that can increase or decrease dinamically (like the memory used by a worker)

ABSOLUTE (type 2)

this is an absolute number, like the memory of the whole server, or the size of the hard disk.

ALIAS (type 3)

this is a virtual metric pointing to another one (you can use it to give different names to already existent metrics)

6.22.3 Metric collectors

Once you define a metric type, you need to tell uWSGI how to 'collect' the specific metric.

There are various ''collectors" available (and new can be added via plugins)

"ptr"

the value is collected from a memory pointer

“file”

the value is collected from a file

“sum”

the value is the sum of other metrics

“avg”

added in 1.9.20

compute the math average of the children

“accumulator”

always add the sum of children to the final value.

Ex:

round1: child1 = 22, child2 = 17 -> metric_value = 39

round2: child1 = 26, child2 = 30 -> metric_value += 56

‘multiplier”

multiply the sum of children for the specified arg1n

child1 = 22, child2 = 17, arg1n = 3 -> metric_value = (22+17)*3

“func”

the value is computed calling a specific function every time

“manual” (the NULL collector)

the value must be updated manually from applications using the metrics api

6.22.4 Custom metrics

You can define additional metrics you can manage from your app.

The `--metric` option allows you to add more metrics.

It has a double syntax: simplified and keyval

```
uwsgi --http-socket :9090 --metric foobar
```

will create a metric ‘foobar’ with type ‘counter’, manual collector and no oid.

For creating advanced metrics you need the keyval way:

```
uwsgi --http-socket :9090 --metric name=foobar,type=gauge,oid=100.100.100
```

The following keys are available:

`name` set the metric name

`oid` set the metric oid

`type` set the metric type, can be `counter`, `gauge`, `absolute`, `alias`

`initial_value` set the metric to a specific value on startup

`freq` set the collection frequency in seconds (default to 1)

`children` maps children to the metric (see below)

`alias` the metric will be a simple alias for the specified one (–metric name=foobar,alias=worker.0.requests,type=alias)

`arg1` .. `arg3` string based arguments (see below)

`arg1n` .. `arg3n` number based arguments (see below)

`collector` set the collector, can be `ptr`, `file`, `sum`, `func` or anything exposed by plugins. Not specifying a collector means the metric is manual (your app needs to update it).

The ptr is currently unimplemented, while the other collector requires a bit of additional configuration:

`collector=file` requires `arg1` for the filename and an optional `arg1n` for the so-called split value.

```
uwsgi --metric name=loadavg,type=gauge,collector=file,arg1=/proc/loadavg,arg1n=1,freq=3
```

this will add a 'loadavg' metric, of type gauge, updated every 3 seconds with the content of /proc/loadavg. The content is splitted (using \n, \t, spaces, \r and zero as separator) and the item 1 (the returned array is zero-based) used as value.

the splitter is very powerful, so you could gather infos from more complex files, like /proc/meminfo

```
uwsgi --metric name=memory,type=gauge,collector=file,arg1=/proc/meminfo,arg1n=4,freq=3
```

once splitted, the /proc/meminfo has the MemFree value in the 4th slot

`collector=sum` requires the list of metrics that must be summed up. Each metric has the concept of 'children'. The sum collector will sum the values of all of its children:

```
uwsgi --metric name=reqs,collector=sum,children=worker.1.requests;worker.2.requests
```

this will sum the value of worker.1.requests and worker.2.requests every second

`collector=func` is a commodity collector avoiding you to write a whole plugin for adding a new collector.

Let's define a C function (call the file mycollector.c or whatever you want):

```
int64_t my_collector(void *metric) {
        return 173;
}
```

and build it as a shared library

```
gcc -shared -o mycollector.so mycollector.c
```

now run uWSGI

```
uwsgi --dlopen ./mycollector.so --metric name=mine,collector=func,arg1=my_collector,freq=10
```

this will call the C function my_collector every 10 seconds and will set the value of the metric 'mine' to its return value.

The function must returns an int64_t value. The argument it takes is a uwsgi_metric pointer. You generally do not need to parse it, so casting to void will avoid headaches.

6.22.5 The metrics directory

UNIX sysadmins love text files. They are generally the things they have to work on most of the time. If you want to make a UNIX sysadmin happy, just give him some text file to play with.

The metrics subsystem can expose all of its metrics in the form of text files in a directory:

```
uwsgi --metrics-dir mymetrics ...
```

(the mymetric dir must exists)

this will create a text file for each metric in the 'mymetrics' directory. The content of each file is the value of the metric (updated in realtime).

Each file is mapped in the process address space, so do not worry if your virtual memory increases.

6.22.6 Restoring metrics (persistent metrics)

When you restart a uWSGI instance, all of its metrics are reset.

This is generally the best thing to do, but if you want you can restore the previous situation, abusing the values stored in the metrics directory defined before.

Just add the `--metrics-dir-restore` option to force the metric subsystem to read-back the values from the metric directory before starting collecting values.

6.22.7 API

Your language plugins should expose at least the following api functions. Currently they are implemented in Perl, CPython, PyPy and Ruby

metric_get(name)

metric_set(name, value)

metric_inc(name[, delta])

metric_dec(name[, delta])

metric_mul(name[, delta])

metric_div(name[, delta])

metrics (tuple/array of metric keys, should be immutable and not-callable, currently unimplemented)

6.22.8 Stats pushers

Collected metrics can be sent to external systems for analysis or graphs generation.

Stats pushers are plugins aimed at sending metrics to those systems.

There are two kinds of stats pusher: json and raw.

json stats pusher send the whole json stats blob (the same you get from the stats server), while 'raw' ones send the metrics list.

Currently available stats pushers:

rrdtool

type: raw

plugin: rrdtool (builtin by default)

requires: librrd.so (dynamically detected on startup, not needed when building)

this will store an rrd file for each metric in the specified directory. Eacch rrd file has a single data source named 'metric'

Usage:

```
uwsgi --rrdtool my_rrds ...
```

or

```
uwsgi --stats-push rrdtool:my_rrds ...
```

by default the rrd files are updated every 300 seconds, you can tune this value with `--rrdtool-freq`

The librrd.so library is detected at runtime. If you need you can specify its absolute path with `--rrdtool-lib`

statsd

type: raw

plugin: stats_pusher_statsd

push metrics to a statsd server

syntax: –stats-push statsd:address[,prefix]

example:

```
uwsgi --stats-push statsd:127.0.0.1:8125,myinstance ...
```

carbon

type: raw

plugin: carbon (builtin by default)

see *Integration with Graphite/Carbon*

zabbix

type: raw

plugin: zabbix

push metrics to a zabbix server

syntax: –stats-push zabbix:address[,prefix]

example:

```
uwsgi --stats-push zabbix:127.0.0.1:10051,myinstance ...
```

The plugin exposes a `--zabbix-template` option that will generate a zabbix template (on stdout or in the specified file) containing all of the exposed metrics as trapper items.

Note: on some zabbox version you need to authorize the ip addresses allowed to push items

mongodb

type: json

plugin: stats_pusher_mongodb

required: libmongoclient.so

push statistics (as json) the the specified mongodb database

syntax (keyval): –stats-push mongodb:addr=<addr>,collection=<db>,freq=<freq>

file

type: json

plugin: stats_pusher_file

example plugin storing stats json in a file

socket

type: raw

plugin: stats_pusher_socket (builtin by default)

push metrics to a udp server with the following format:

<metric> <type> <value>

(<type> is in the numeric form previously reported)

syntax: –stats-push socket:address[,prefix]

Example:

```
uwsgi --stats-push socket:127.0.0.1:8125,myinstance ...
```

6.22.9 Alarms/Thresholds

You can configure one or more “thresholds” to each metric.

Once this limit is reached the specified alarm (see *The uWSGI alarm subsystem (from 1.3)*) is triggered.

Once the alarm is delivered you may choose to reset the counter to aspecfic value (generally 0), or continue triggering alarms with a specified rate.

```
[uwsgi]
...
metric-alarm = key=worker.0.avg_response_time,value=2000,alarm=overload,rate=30
metric-alarm = key=loadavg,value=3,alarm=overload,rate=120
metric-threshold = key=mycounter,value=1000,reset=0
...
```

Specifying an alarm is not required, using the threshold value to automatically reset a metric is perfectly valid

Note: –metric-threshold and –metric-alarm are the same option

6.22.10 SNMP integration

The *The embedded SNMP server* server exposes metrics starting from the 1.3.6.1.4.1.35156.17.3 OID.

For example to get the value of worker.0.requests:

```
snmpget -v2c -c <snmp_community> <snmp_addr>:<snmp_port> 1.3.6.1.4.1.35156.17.3.0.1
```

Remember: only metrics with an associated OID can be used via SNMP

6.22.11 Internal Routing integration

The ''router_metrics" plugin (builtin by default) adds a series of actions to the internal routing subsystem.

`metricinc:<metric>[,value]` increase the <metric>

`metricdec:<metric>[,value]` decrease the <metric>

`metricmul:<metric>[,value]` multiply the <metric>

`metricdiv:<metric>[,value]` divide the <metric>

`metricset:<metric>,<value>` set <metric> to <value>

in addition to action a route var named "metric" is added

Example:

```
[uwsgi]
metric = mymetric
route = ^/foo metricinc:mymetric
route-run = log:the value of the metric 'mymetric' is ${metric[mymetric]}
log-format = %(time) - %(metric.mymetric)
```

6.22.12 Request logging

You can access metrics values from your request logging format using the %(metric.xxx) placeholder:

```
[uwsgi]
log-format = [hello] %(time) %(metric.worker.0.requests)
```

6.22.13 Officially Registered Metrics

This is a work in progress, best way to know which default metrics are exposed is enabling the stats server and querying it (or adding the –metrics-dir option)

- worker/3 (exports information about workers, example worker.1.requests [or 3.1.1] reports the number of requests served by worker 1)
- plugin/4 (namespace for metrics automatically added by plugins, example plugins.foo.bar)
- core/5 (namespace for general instance informations)
- router/6 (namespace for corerouters, example router.http.active_sessions)

- socket/7 (namespace for sockets, example socket.0.listen_queue)
- mule/8 (namespace for mules, example mule.1.signals)
- spooler/9 (namespace for spoolers, example spooler.1.signals)
- system/10 (namespace for system metrics, like loadavg or free memory)

6.22.14 OID assigment for plugins

If you want to write plugin that will expose metrics, please first add OID namespace that you are going to use to the list below and make a pull request. This will ensure that all plugins are using unique OID namespaces. Prefix all plugin metric names with plugin name to ensure no conflicts if same keys are used in multiple plugins (example plugin.myplugin.foo.bar, worker.1.plugin.myplugin.foo.bar)

- (3|4).100.1 - cheaper_busyness

6.22.15 External tools

Check https://github.com/unbit/unbit-bars

6.23 The Chunked input API

An api for managing HTTP chunked input requests has been added in uWSGI 1.9.13

The api is very low-level to allows easy integration with standard apps.

There are only two functions exposed:

- chunked_read([timeout])
- chunked_read_nb()

The api is supported (as uWSGI 1.9.20) on CPython, PyPy and Perl

6.23.1 Reading chunks

To read a chunk (in blocking way) just run

```
my $msg = uwsgi::chunked_read
```

if no timeout is specified the default one will be used, so if you do not get a chunk in time, the function will croak (or will raise an exception when under python).

Under non-blocking/async engines you may want to use

```
my $msg = uwsgi::chunked_read_nb
```

the function will soon return 'undef' (or None on python) if no chunks are available (and will croak/raise an exception on error)

A full PSGI streaming echo example:

```
# simple PSGI echo app reading chunekd input
sub streamer {
     $responder = shift;
     # generate the headers and start streaming the response
     my $writer = $responder->( [200, ['Content-Type' => 'text/plain']]);

     while(1) {
             my $msg = uwsgi::chunked_read;
             last unless $msg;
             $writer->write($msg);
     }

     $writer->close;
}

my $app = sub {
     return \&streamer;
};
```

6.23.2 Tuning the chunks buffer

Before start reading chunks, uWSGI allocates a fixed buffer for storing chunks.

All of the messages are always stored in the same buffer. If a message bigger than the buffer is received an exception will be raised.

By default the buffer is limited to 1MB, you can tune it with the `--chunked-input-limit` option (it takes bytes)

6.23.3 Integration with proxies

If you plan to put uWSGI behind a proxy/router be sure it supports chunked input requests (or generally raw HTTP requests).

When using the uWSGI http router just add –http-raw-body to support chunked input

HAProxy works out of the box

nginx >= 1.4 supports chunked input

6.23.4 Options

`--chunked-input-limit` the limit (in bytes) of a chunk message (default 1MB)

`--chunked-input-timeout` the default timeout (in seconds) for blocking chunked_read (default to the same –socket-timeout value, 4 seconds)

6.23.5 Notes

Calling chunked api functions after having consumed even a single byte of the request body is wrong (this includes post buffering)

Chunked api function can be called independently by the presence of “Transfer-Encoding: chunked” header

CHAPTER 7

Scaling with uWSGI

7.1 The uWSGI cheaper subsystem – adaptive process spawning

uWSGI provides the ability to dynamically scale the number of running workers via pluggable algorithms. Use `uwsgi --cheaper-algos-list` to get the list of available algorithms.

7.1.1 Usage

To enable cheaper mode add the `cheaper = N` option to the uWSGI configuration file, where N is the minimum number of workers uWSGI can run. The `cheaper` value must be lower than the maximum number of configured workers (`workers` or `processes` option).

```
# set cheaper algorithm to use, if not set default will be used
cheaper-algo = spare

# minimum number of workers to keep at all times
cheaper = 2

# number of workers to spawn at startup
cheaper-initial = 5

# maximum number of workers that can be spawned
workers = 10

# how many workers should be spawned at a time
cheaper-step = 1
```

This configuration will tell uWSGI to run up to 10 workers under load. If the app is idle uWSGI will stop workers but it will always leave at least 2 of them running. With `cheaper-initial` you can control how many workers should be spawned at startup. If your average load requires more than minimum number of workers you can have them spawned right away and then "cheaped" (killed off) if load is low enough. When the cheaper algorithm decides that it needs more workers it will spawn `cheaper-step` of them. This is useful if you have a high maximum number of workers – in the event of a sudden load spike it would otherwise take a lot of time to spawn enough workers one by one.

7.1.2 Setting memory limits

Starting with 1.9.16 rss memory limits can be set to stop cheaper spawning new workers if process count limit was not reached, but total sum of rss memory used by all workers reached given limit.

```
# soft limit will prevent cheaper from spawning new workers
# if workers total rss memory is equal or higher
# we use 128MB soft limit below (values are in bytes)
cheaper-rss-limit-soft = 134217728

# hard limit will force cheaper to cheap single worker
# if workers total rss memory is equal or higher
# we use 160MB hard limit below (values are in bytes)
cheaper-rss-limit-hard = 167772160
```

Notes:

- Hard limit is optional, soft limit alone can be used.
- Hard value must be higher then soft value, both values shouldn't be too close to each other.
- Hard value should be soft + at least average worker memory usage for given app.
- Soft value is the limiter for cheaper, it won't spawn more workers, but already running workers memory usage might grow, to handle that reload-on-rss can be set to. To set unbreakable barrier for app memory usage cgroups are recommended.

7.1.3 `spare` cheaper algorithm

This is the default algorithm. If all workers are busy for `cheaper_overload` seconds then uWSGI will spawn new workers. When the load is gone it will begin stopping processes one at a time.

7.1.4 `backlog` cheaper algorithm

Note: `backlog` is only available on Linux and only on TCP sockets (not UNIX domain sockets).

If the socket's listen queue has more than `cheaper_overload` requests waiting to be processed, uWSGI will spawn new workers. If the backlog is lower it will begin killing processes one at a time.

7.1.5 `cheaper` busyness algorithm

Note: This algorithm is optional, it is only available if the `cheaper_busyness` plugin is compiled and loaded.

This plugin implements an algorithm which adds or removes workers based on average utilization for a given time period. It's goal is to keep more workers than the minimum needed available at any given time, so the app will always have capacity for new requests. If you want to run only minimum number of workers then use the spare or backlog algorithms.

This plugin primarily is used because the way spare and backlog plugins work causes very aggressive scaling behavior. If you set a low `cheaper` value (for example 1), then uWSGI will keep only 1 worker running and spawn new workers only when that running worker is overloaded. If an app requires more workers, then uWSGI will be spawning and stopping workers all the time. Only during times of very low load the would the minimum number of workers be enough.

The Busyness algorithm tries to do the opposite: spawn as many workers as needed and stop some of them only when there is a good chance that they are not needed. This should lead to a more stable worker count and much less respawns. Since for most of the time we have more worker capacity than actually needed, average application response times should be lower than with other plugins.

Options:

cheaper-overload

Specifies the window, in seconds, for tracking the average busyness of workers. Example:

```
cheaper-overload = 30
```

This option will check busyness every 30 seconds. If during the last 30 seconds all workers were running for 3 seconds and idle for the remaining 27 seconds the calculated busyness will be 10% (3/30). This value will decide how fast uWSGI can respond to load spikes. New workers will be spawned at most every `cheaper-overload` seconds (unless you are running uWSGI on Linux – see `cheaper-busyness-backlog-alert` for details).

If you want to react to load spikes faster, keep this value low so busyness is calculated more often. Keep in mind this may cause workers to be started/stopped more often than required since every minor spike may spawn new workers. With a high `cheaper-overload` value the worker count will change much less since longer cycles will eat all short spikes of load and extreme values.

cheaper-step

How many workers to spawn when the algorithm decides they are needed. Default is 1.

cheaper-initial

The number of workers to be started when starting the application. After the app is started the algorithm can stop or start workers if needed.

cheaper-busyness-max

This is the maximum busyness we allow. Every time the calculated busyness for last `cheaper-overload` seconds is higher than this value, uWSGI will spawn `cheaper-step` new workers. Default is 50.

cheaper-busyness-min

This is minimum busyness. If current busyness is below this value, the app is considered as being in an "idle cycl" and uWSGI will start counting them. Once we reach needed number of idle cycles uWSGI will kill one worker. Default is 25.

cheaper-busyness-multiplier

This option tells uWSGI how many idle cycles we need before stopping a worker. After reaching this limit uWSGI will stop a worker and reset this counter.

For example:

```
cheaper-overload = 10
cheaper-busyness-multiplier = 20
cheaper-busyness-min = 25
```

If average worker busyness is under 25% for 20 checks in a row, executed every 10 seconds (total of 200 seconds), tone worker will be stopped. The idle cycles counter will be reset if average busyness jumps above `cheaper-busyness-max` and we spawn new workers. If during idle cycle counting the average busyness jumps above `cheaper-busyness-min` but still below `cheaper-busyness-max`, then the idle cycles counter is adjusted and we need to wait extra one idle cycle. If during idle cycle counting the average busyness jumps above `cheaper-busyness-min` but still below `cheaper-busyness-max` three times in a row, then the idle cycle counter is reset.

cheaper-busyness-penalty

uWSGI will automatically tune the number of idle cycles needed to stop worker when worker is stopped due to enough idle cycles and then spawned back to fast (less than the same time we need to cheap worker), then we will increment the `cheaper-busyness-multiplier` value this value. Default is 1.

Example:

```
cheaper-overload = 10
cheaper-busyness-multiplier = 20
cheaper-busyness-min = 25
cheaper-busyness-penalty = 2
```

If average worker busyness is under 25% for 20 checks in a row, executed every 10 seconds (total 200 seconds), one worker will be stopped. If new worker is spawned in less than 200 seconds (counting from the time when we spawned the last worker before it), the `cheaper-busyness-multiplier` value will be incremented up to 22 (20+2). Now we will need to wait 220 seconds (22*10) to cheap another worker. This option is used to prevent workers from being started and stopped all the time since once we stop one worker, busyness might jump up enough to hit `cheaper-busyness-max`. Without this, or if tuned poorly, we can get into a stop/start feedback loop .

cheaper-busyness-verbose

This option enables debug logs from the `cheaper_busyness` plugin.

cheaper-busyness-backlog-alert

This option is only available on Linux. It is used to allow quick response to load spikes even with high `cheaper-overload` values. On every uWSGI master cycle (default 1 second) the current listen queue is checked. If it is higher than this value, an emergency worker is spawned. When using this option it is safe to use high `cheaper-overload` values to have smoother scaling of worker count. Default is 33.

cheaper-busyness-backlog-multiplier

This option is only available on Linux. It works just like `cheaper-busyness-multiplier`, except it is used only for emergency workers spawned when listen queue was higher than `cheaper-busyness-backlog-alert`.

Emergency workers are spawned in case of big load spike to prevent currently running workers from being overloaded. Sometimes load spike are random and short which can spawn a lot of emergency workers. In such cases we would need to wait several cycles before reaping those workers. This provides an alternate multiplier to reap these processes faster. Default is 3.

cheaper-busyness-backlog-step

This option is only available on Linux. It sets the number of emergency workers spawned when listen queue is higher than `cheaper-busyness-backlog-alert`. Defaults to 1.

cheaper-busyness-backlog-nonzero

This option is only available on Linux. It will spawn new emergency workers if the request listen queue is > 0 for more than N seconds. It is used to protect the server from the corner case where there is only a single worker running and the worker is handling a long running request. If uWSGI receives new requests they would stay in the request queue until that long running request is completed. With this option we can detect such a condition and spawn new worker to prevent queued requests from being timed out. Default is 60.

Notes regarding Busyness

- Experiment with settings, there is no one golden rule of what values should be used for everyone. Test and pick values that are best for you. Monitoring uWSGI stats (via Carbon, for instance) will make it easy to decide on good values.
- Don't expect busyness to be constant. it will change frequently. In the end, real users interact with your apps in very random way. It's recommended to use longer –cheaper-overload values (>=30) to have less spikes.
- If you want to run some benchmarks with this plugin, you should use tools that add randomness to the work load
- With a low number of workers (2-3) starting new worker or stopping one might affect busyness a lot, if You have 2 workers with busyness of 50%, than stopping one of them will increase busyness to 100%. Keep that in mind when picking min and max levels, with only few workers running most of the time max should be more than double of min, otherwise every time one worker is stopped it might increase busyness to above max level.
- With a low number of workers (1-4) and default settings expect this plugin will keep average busyness below the minimum level; adjust levels to compensate for this.
- With a higher number of workers required to handle load, worker count should stabilize somewhere near minimum busyness level, jumping a little bit around this value
- When experimenting with this plugin it is advised to enable `--cheaper-busyness-verbose` to get an idea of what it is doing. An example log follows.

7.2 The uWSGI Emperor – multi-app deployment

If you need to deploy a big number of apps on a single server, or a group of servers, the Emperor mode is just the ticket. It is a special uWSGI instance that will monitor specific events and will spawn/stop/reload instances (known as *vassals*, when managed by an Emperor) on demand.

By default the Emperor will scan specific directories for supported (.ini, .xml, .yml, .json, etc.) uWSGI configuration files, but it is extensible using *imperial monitor* plugins. The `dir://` and `glob://` plugins are embedded in the core, so they need not be loaded, and are automatically detected. The `dir://` plugin is the default.

- Whenever an imperial monitor detects a new configuration file, a new uWSGI instance will be spawned with that configuration.
- Whenever a configuration file is modified (its modification time changed, so `touch` may be your friend), the corresponding app will be reloaded.
- Whenever a config file is removed, the corresponding app will be stopped.
- If the emperor dies, all the vassals die.
- If a vassal dies for any reason, the emperor will respawn it.

Multiple sources of configuration may be monitored by specifying `--emperor` multiple times.

See also:

See *Imperial monitors* for a list of the Imperial Monitor plugins shipped with uWSGI and how to use them.

7.2.1 Imperial monitors

`dir://` – scan a directory for uWSGI config files

Simply put all of your config files in a directory, then point the uWSGI emperor to it. The Emperor will start scanning this directory. When it finds a valid config file it will spawn a new uWSGI instance.

For our example, we're deploying a Werkzeug test app, a Trac instance, a Ruby on Rails app and a Django app.

werkzeug.xml

```
<uwsgi>
    <module>werkzeug.testapp:test_app</module>
    <master/>
    <processes>4</processes>
    <socket>127.0.0.1:3031</socket>
</uwsgi>
```

trac.ini

```
[uwsgi]
master = true
processes = 2
module = trac.web.main:dispatch_request
env = TRAC_ENV=/opt/project001
socket = 127.0.0.1:3032
```

rails.yml

```
uwsgi:
    plugins: rack
    rack: config.ru
    master: 1
    processes: 8
    socket: 127.0.0.1:3033
    post-buffering: 4096
    chdir: /opt/railsapp001
```

django.ini

```
[uwsgi]
socket = 127.0.0.1:3034
threads = 40
master = 1
env = DJANGO_SETTINGS_MODULE=myapp.settings
module = django.core.handlers.wsgi:WSGIHandler()
chdir = /opt/djangoapp001
```

Put these 4 files in a directory, for instance `/etc/uwsgi/vassals` in our example, then spawn the Emperor:

```
uwsgi --emperor /etc/uwsgi/vassals
```

The emperor will find the uWSGI instance configuration files in that directory (the `dir://` plugin declaration is implicit) and start the daemons needed to run them.

glob:// – monitor a shell pattern

glob:// is similar to dir://, but a glob expression must be specified:

```
uwsgi --emperor "/etc/vassals/domains/*/conf/uwsgi.xml"
uwsgi --emperor "/etc/vassals/*.ini"
```

Note: Remember to quote the pattern, otherwise your shell will most likely interpret it and expand it at invocation time, which is not what you want.

As the Emperor can search for configuration files in subdirectory hierarchies, you could have a structure like this:

```
/opt/apps/app1/app1.xml
/opt/apps/app1/...all the app files...
/opt/apps/app2/app2.ini
/opt/apps/app2/...all the app files...
```

and run uWSGI with:

```
uwsgi --emperor /opt/apps/app*/app*.*
```

pg:// – scan a PostgreSQL table for configuration

You can specify a query to run against a PostgreSQL database. Its result must be a list of 3 to 5 fields defining a vassal:

1. The instance name, including a valid uWSGI config file extension. (Such as django-001.ini)
2. A TEXT blob containing the vassal configuration, in the format based on the extension in field 1
3. A number representing the modification time of this row in UNIX format (seconds since the epoch).
4. The UID of the vassal instance. Required in *Tyrant mode (secure multi-user hosting)* mode only.
5. The GID of the vassal instance. Required in *Tyrant mode (secure multi-user hosting)* mode only.

```
uwsgi --plugin emperor_pg --emperor "pg://host=127.0.0.1 user=foobar dbname=emperor;SELECT name,confi
```

- Whenever a new tuple is added a new instance is created and spawned with the config specified in the second field.
- Whenever the modification time field changes, the instance is reloaded.
- If a tuple is removed, the corresponding vassal will be destroyed.

mongodb:// – Scan MongoDB collections for configuration

```
uwsgi --plugin emperor_mongodb --emperor "mongodb://127.0.0.1:27107,emperor.vassals,{enabled:1}"
```

This will scan all of the documents in the emperor.vassals collection having the field enabled set to 1. An Emperor-compliant document must define three fields: name, config and ts. In *Tyrant mode (secure multi-user hosting)* mode, 2 more fields are required.

- name (string) is the name of the vassal (remember to give it a valid extension, like .ini)
- config (multiline string) is the vassal config in the format described by the name's extension.
- ts (date) is the timestamp of the config (Note: MongoDB internally stores the timestamp in milliseconds.)
- uid (number) is the UID to run the vassal as. Required in *Tyrant mode (secure multi-user hosting)* mode only.

- `gid` (number) is the GID to run the vassal as. Required in *Tyrant mode (secure multi-user hosting)* mode only.

`amqp://` – Use an AMQP compliant message queue to announce events

Set your AMQP (RabbitMQ, for instance) server address as the –emperor argument:

```
uwsgi --plugin emperor_amqp --emperor amqp://192.168.0.1:5672
```

Now the Emperor will wait for messages in the `uwsgi.emperor` exchange. This should be a *fanout* type exchange, but you can use other systems for your specific needs. Messages are simple strings containing the absolute path of a valid uWSGI config file.

```
# The pika module is used in this example, but you're free to use whatever adapter you like.
import pika
# connect to RabbitMQ server
connection = pika.BlockingConnection(pika.ConnectionParameters('192.168.0.1'))
# get the channel
channel = connection.channel()
# create the exchange (if not already available)
channel.exchange_declare(exchange='uwsgi.emperor', type='fanout')
# publish a new config file
channel.basic_publish(exchange='uwsgi.emperor', routing_key='', body='/etc/vassals/mydjangoapp.xml')
```

The first time you launch the script, the emperor will add the new instance (if the config file is available). From now on every time you re-publish the message the app will be reloaded. When you remove the config file the app is removed too.

Tip: You can subscribe all of your emperors in the various servers to this exchange to allow cluster-synchronized reloading/deploy.

AMQP with HTTP

uWSGI *is capable of loading configuration files over HTTP*. This is a very handy way to dynamically generate configuration files for massive hosting. Simply declare the HTTP URL of the config file in the AMQP message. Remember that it must end with one of the valid config extensions, but under the hood it can be generated by a script. If the HTTP URL returns a non-200 status code, the instance will be removed.

```
channel.basic_publish(exchange='uwsgi.emperor', routing_key='', body='http://example.com/confs/trac.i
```

Direct AMQP configuration

Configuration files may also be served directly over AMQP. The `routing_key` will be the (virtual) config filename, and the message will be the content of the config file.

```
channel.basic_publish(
  exchange='uwsgi.emperor',
  routing_key='mydomain_trac_config.ini',
  body="""
[uwsgi]
socket=:3031
env = TRAC_ENV=/accounts/unbit/trac/uwsgi
module = trac.web.main:dispatch_request
processes = 4""")
```

The same reloading rules of previous modes are valid. When you want to remove an instance simply set an empty body as the "configuration".

```
channel.basic_publish(exchange='uwsgi.emperor', routing_key='mydomain_trac_config.ini', body='')
```

zmq:// – ZeroMQ

The Emperor binds itself to a ZeroMQ PULL socket, ready to receive commands.

```
uwsgi --plugin emperor_zeromq --emperor zmq://tcp://127.0.0.1:5252
```

Each command is a multipart message sent over a PUSH zmq socket. A command is composed by at least 2 parts: `command` and `name` `command` is the action to execute, while `name` is the name of the vassal. 3 optional parts can be specified.

- `config` (a string containing the vassal config)
- `uid` (the user id to drop priviliges to in case of tyrant mode)
- `gid` (the group id to drop priviliges to in case of tyrant mode)

There are 2 kind of commands (for now):

- `touch`
- `destroy`

The first one is used for creating and reloading instances while the second is for destroying. If you do not specify a config string, the Emperor will assume you are referring to a static file available in the Emperor current directory.

```
import zmq
c = zmq.Context()
s = zmq.Socket(c, zmq.PUSH)
s.connect('tcp://127.0.0.1:5252')
s.send_multipart(['touch','foo.ini',"[uwsgi]\nsocket=:4142"])
```

zoo:// – Zookeeper

Currently in development.

ldap:// – LDAP

Currently in development.

7.2.2 The Emperor protocol

As of 1.3 you can spawn custom applications via the *Emperor*.

Non-uWSGI Vassals should never daemonize, to maintain a link with the Emperor. If you want/need better integration with the Emperor, implement the Emperor protocol.

The protocol

An environment variable `UWSGI_EMPEROR_FD` is passed to every vassal, containing a file descriptor number.

```
import os
has_emperor = os.environ.get('UWSGI_EMPEROR_FD')
if has_emperor:
   print "I'm a vassal snake!"
```

Or in Perl,

```
my $has_emperor = $ENV{'UWSGI_EMPEROR_FD'}
if ($has_emperor) {
  print "I am a vassal.\n"
}
```

Or in C,

```
int emperor_fd = -1;
char *has_emperor = getenv("UWSGI_EMPEROR_FD");
if (has_emperor) {
    emperor_fd = atoi(has_emperor);
    fprintf(stderr, "I am a vassal.\n");
}
```

From now you can receive (and send) messages from (and to) the Emperor over this file descriptor.

Messages are byte sized (0-255), and each number (byte) has a meaning.

0	Sent by the Emperor to stop a vassal
1	Sent by the Emperor to reload a vassal / sent by a vassal when it has been spawned
2	Sent by a vassal to ask the Emperor for configuration chunk
5	Sent by a vassal when it is ready to accept requests
17	Sent by a vassal after the first request to announce loyalty
22	Sent by a vassal to notify the Emperor of voluntary shutdown
26	Heartbeat sent by the vassal. After the first received heartbeat, the Emperor will expect more of them from the vassal.
30	Sent by the vassal to ask for *Auto-scaling with Broodlord mode* mode.

7.2.3 Special configuration variables

Using *Placeholders* and *Magic variables* in conjunction with the Emperor will probably save you a lot of time and make your configuration more DRY. Suppose that in /opt/apps there are only Django apps. /opt/apps/app.skel (the .skel extension is not a known configuration file type to uWSGI and will be skipped)

```
[uwsgi]
chdir = /opt/apps/%n
master = true
threads = 20
socket = /tmp/sockets/%n.sock
env = DJANGO_SETTINGS_MODULE=%n.settings
module = django.core.handlers.wsgi:WSGIHandler()
```

And then for each app create a symlink:

```
ln -s /opt/apps/app.skel /opt/apps/app1.ini
ln -s /opt/apps/app.skel /opt/apps/app2.ini
```

7.2.4 Passing configuration parameters to all vassals

Starting from 1.9.19 you can pass options using the `--vassal-set` facility

```
[uwsgi]
emperor = /etc/uwsgi/vassals
vassal-set = processes=8
vassal-set = enable-metrics=1
```

this will add `--set processes=8` and `--set enable-metrics=1` to each vassal

You can force the Emperor to pass options to uWSGI instances using environment variables too. Every environment variable of the form `UWSGI_VASSAL_xxx` will be rewritten in the new instance as `UWSGI_xxx`, with the usual *configuration implications*.

For example:

```
UWSGI_VASSAL_SOCKET=/tmp/%n.sock uwsgi --emperor /opt/apps
```

will let you avoid specifying the socket option in configuration files.

Alternatively, you can use the `--vassals-include` option let each vassal automatically include a complete config file:

```
uwsgi --emperor /opt/apps --vassals-include /etc/uwsgi/vassals-default.ini
```

Note that if you do this, `%n` (and other magic variables) in the included file will resolve to the name of the included file, not the original vassal configuration file. If you want to set options in the included file using the vassal name, you'll have to use placeholders. For example, in the vassal config, you write:

```
[uwsgi]
vassal_name = %n
... more options
```

In the `vassal-defaults.ini`, you write:

```
[uwsgi]
socket = /tmp/sockets/%(vassal_name).sock
```

7.2.5 Tyrant mode (secure multi-user hosting)

The emperor is normally be run as root, setting the UID and GID in each instance's config. The vassal instance then drops privileges before serving requests. In this mode, if your users have access to their own uWSGI configuration files, you can't trust them to set the correct `uid` and `gid`. You could run the emperor as unprivileged user (with `uid` and `gid`) but all of the vassals would then run under the same user, as unprivileged users are not able to promote themselves to other users. For this case the Tyrant mode is available – just add the `emperor-tyrant` option.

In Tyrant mode the Emperor will run the vassal with the UID/GID of its configuration file (or for other Imperial Monitors, by some other method of configuration). If Tyrant mode is used, the vassal configuration files must have UID/GID > 0. An error will occur if the UID or GID is zero, or if the UID or GID of the configuration of an already running vassal changes.

Tyrant mode for paranoid sysadmins (Linux only)

If you have built a uWSGI version with *Setting POSIX Capabilities* options enabled, you can run the Emperor as unprivileged user but maintaining the minimal amount of root-capabilities needed to apply the tyrant mode

```
[uwsgi]
uid = 10000
gid = 10000
emperor = /tmp
emperor-tyrant = true
cap = setgid,setuid
```

7.2.6 On demand vassals (socket activation)

Inspired by the venerable xinetd/inetd approach, you can spawn your vassals only after the first connection to a specific socket. This feature is available as of 1.9.1. Check the changelog for more information: *uWSGI 1.9.1*

7.2.7 Loyalty

As soon as a vassal manages a request it will became "loyal". This status is used by the Emperor to identify bad-behaving vassals and punish them.

7.2.8 Throttling

Whenever two or more vassals are spawned in the same second, the Emperor will start a throttling subsystem to avoid fork bombing. The system adds a throttle delta (specified in milliseconds via the *emperor-throttle* option) whenever it happens, and waits for that duration before spawning a new vassal. Every time a new vassal spawns without triggering throttling, the current throttling duration is halved.

7.2.9 Blacklist system

Whenever a non-loyal vassal dies, it is put in a shameful blacklist. When in a blacklist, that vassal will be throttled up to a maximum value (tunable via *emperor-max-throttle*), starting from the default throttle delta of 3. Whenever a blacklisted vassal dies, its throttling value is increased by the delta (*emperor-throttle*).

7.2.10 Heartbeat system

Vassals can voluntarily ask the Emperor to monitor their status. Workers of heartbeat-enabled vassals will send "heartbeat" messages to the Emperor. If the Emperor does not receive heartbeats from an instance for more than N (default 30, *emperor-required-heartbeat*) seconds, that instance will be considered hung and thus reloaded. To enable sending of heartbeat packet in a vassal, add the *heartbeat* option.

Important: If all of your workers are stuck handling perfectly legal requests such as slow, large file uploads, the Emperor will trigger a reload as if the workers are hung. The reload triggered is a graceful one, so you can be able to tune your config/timeout/mercy for sane behaviour.

7.2.11 Using Linux namespaces for vassals

On Linux you can tell the Emperor to run vassals in "unshared" contexts. That means you can run each vassal with a dedicated view of the filesystems, ipc, uts, networking, pids and uids.

Things you generally do with tools like `lxc` or its abstractions like `docker` are native in uWSGI.

For example if you want to run each vassals in a new namespace:

```
[uwsgi]
emperor = /etc/uwsgi/vassals
emperor-use-clone = fs,net,ipc,pid,uts
```

now each vassal will be able to modify the filesystem layout, networking, hostname and so on without damaging the main system.

A couple of helper daemons are included in the uWSGI distribution to simplify management of jailed vassals. Most notably *The TunTap Router* allows full user-space networking in jails, while the `forkpty router` allows allocation of pseudoterminals in jails

It is not needed to unshare all of the subsystem in your vassals, sometimes you only want to give dedicated ipc and hostname to a vassal and hide from the processes list:

```
[uwsgi]
emperor = /etc/uwsgi/vassals
emperor-use-clone = fs,ipc,pid,uts
```

a vassal could be:

```
[uwsgi]
; set the hostname
exec-as-root = hostname foobar
; umount /proc and remount to hide processes
; as we are in the 'fs' namespace umounting /proc does not interfere with the main one
exec-as-root = umount /proc
exec-as-root = mount -t proc none /proc
; drop privileges
uid = foobar
gid = foobar
; bind to the socket
socket = /tmp/myapp.socket
psgi = myapp.pl
```

7.2.12 The Imperial Bureau of Statistics

You can enable a statistics/status service for the Emperor by adding the *emperor-stats, emperor-stats-server* option with a TCP address. By connecting to that address, you'll get a JSON-format blob of statistics.

7.2.13 Running non-uWSGI apps or using alternative uWSGIs as vassals

You can `exec()` a different binary as your vassal using the `privileged-binary-patch`/`unprivileged-binary-patch` options. The first one patches the binary after socket inheritance and shared socket initialization (so you will be able to use uWSGI-defined sockets). The second one patches the binary after privileges drop. In this way you will be able to use uWSGI's UID/GID/chroot/namespace/jailing options. The binary is called with the same arguments that were passed to the vassal by the Emperor.

```
; i am a special vassal calling a different binary in a new linux network namespace
[uwsgi]
uid = 1000
gid = 1000
unshare = net
unprivileged-binary-patch = /usr/bin/myfunnyserver
```

Important: *DO NOT DAEMONIZE* your apps. If you do so, the Emperor will lose its connection with them.

The uWSGI arguments are passed to the new binary. If you do not like that behaviour (or need to pass custom arguments) add `-arg` to the binary patch option, yielding:

```
; i am a special vassal calling a different binary in a new linux network namespace
; with custom options
[uwsgi]
uid = 1000
gid = 1000
unshare = net
unprivileged-binary-patch-arg = ps aux
```

or:

```
;nginx example
[uwsgi]
privileged-binary-patch-arg = nginx -g "daemon off;"
```

See also:

Your custom vassal apps can also speak with the emperor using the *emperor protocol*.

7.2.14 Integrating the Emperor with the FastRouter

The FastRouter is a proxy/load-balancer/router speaking *The uwsgi Protocol*. Yann Malet from Lincoln Loop has released a draft about massive Emperor + Fastrouter deployment (PDF) using *The uWSGI caching framework* as a hostname to socket mapping storage.

7.2.15 Notes

- At startup, the emperor `chdir()` to the vassal dir. All vassal instances will start from here.
- If the uwsgi binary is not in your system path you can force its path with `binary-path`:

  ```
  ./uwsgi --emperor /opt/apps --binary-path /opt/uwsgi/uwsgi
  ```

- Sending `SIGUSR1` to the emperor will print vassal status in its log.
- Stopping (`SIGINT`/`SIGTERM`/`SIGQUIT`) the Emperor will invoke Ragnarok and kill all the vassals.
- Sending `SIGHUP` to the Emperor will reload all vassals.
- The emperor should generally not be run with `--master`, unless master features like advanced logging are specifically needed.
- The emperor should generally be started at server boot time and left alone, not reloaded/restarted except for uWSGI upgrades; emperor reloads are a bit drastic, reloading all vassals at once. Instead vassals should be reloaded individually when needed, in the manner of the imperial monitor in use.

7.2.16 Todo

- Docs-TODO: Clarify what the "chdir-on-startup" behavior does with non-filesystem monitors.
- Export more magic vars
- Add support for multiple sections in xml/ini/yaml files (this will allow to have a single config file for multiple instances)

7.3 Auto-scaling with Broodlord mode

Broodlord (taken from Starcraft, like *Zerg mode* mode) is a way for vassals to ask for more workers from the Emperor. Broodlord mode alone is not very useful. However, when combined with *Zerg mode*, `Idle` and *The uWSGI Emperor – multi-app deployment* it can be used to implement auto-scaling for your apps.

7.3.1 A simple example

We'll start apps with a single worker, adding resources on demand. Broodlord mode expects an additional stanza in your config file to be used for zergs.

```
[uwsgi]
socket = :3031
master = true
vassal-sos-backlog = 10
module = werkzeug.testapp:test_app
processes = 1
zerg-server = /tmp/broodlord.sock
disable-logging = true

[zerg]
zerg = /tmp/broodlord.sock
master = true
module = werkzeug.testapp:test_app
processes = 1
disable-logging = true
idle = 30
die-on-idle = true
```

The `vassal-sos-backlog` option (supported only on Linux and TCP sockets) will ask the Emperor for zergs when the listen queue is higher than the given value. By default the value is 10. More "vassal-sos-" options will be added in the future to allow for more specific detect-overload systems.

The `[zerg]` stanza is the config the Emperor will run when a vassal requires resources. The `die-on-idle` option will completely destroy the zerg when inactive for more than 30 seconds. This configuration shows how to combine the various uWSGI features to implement different means of scaling. To run the Emperor we need to specify how many zerg instances can be run:

```
uwsgi --emperor /etc/vassals --emperor-broodlord 40
```

This will allow you to run up to 40 additional zerg workers for your apps.

7.4 Zerg mode

Note: Yes, that's Zerg as in the "quantity-over-quality" Starcraft race. If you haven't played Starcraft, be prepared for some nonsense.

Note: Also note that this nonsense is mostly limited to the nomenclature. Zerg Mode is serious business.

When your site load is variable, it would be nice to be able to add workers dynamically.

You can obviously edit your configuration to hike up `workers` and reload your uWSGI instance, but for very loaded apps this is undesirable, and frankly – who wants to do manual work like that to scale an app?

Enabling Zerg mode you can allow "uwsgi-zerg" instances to attach to your already running server and help it in the work.

Zerg mode is obviously local only. You cannot use it to add remote instances – this is a job better done by the *The uWSGI FastRouter*, the *HTTP plugin* or your web server's load balancer.

7.4.1 Enabling the zerg server

If you want an uWSGI instance to be rushed by zerg, you have to enable the Zerg server. It will be bound to an UNIX socket and will pass uwsgi socket file descriptors to the Zerg workers connecting to it.

> **Warning:** The socket must be an UNIX socket because it must be capable of passing through file descriptors. A TCP socket simply will not work.

For security reasons the UNIX socket does not inherit the `chmod-socket` option, but will always use the current umask.

If you have filesystem permission issues, on Linux you can use the UNIX sockets in abstract namespace, by prepending an `@` to the socket name.

- A normal UNIX socket:

```
./uwsgi -M -p 8 --module welcome --zerg-server /var/run/mutalisk
```

- A socket in a Linux abstract namespace:

```
./uwsgi -M -p 8 --module welcome --zerg-server @nydus
```

7.4.2 Attaching zergs to the zerg server

To add a new instance to your zerg pool, simply use the –zerg option

```
./uwsgi --zerg /var/run/mutalisk --master --processes 4 --module welcome
# (or --zerg @nydus, following the example above)
```

In this way 4 new workers will start serving requests.

When your load returns to normal values, you can simply shutdown all of the uwsgi-zerg instances without problems.

You can attach an unlimited number of uwsgi-zerg instances.

7.4.3 Fallback if a zerg server is not available

By default a Zerg client will not run if the Zerg server is not available. Thus, if your zerg server dies, and you reload the zerg client, it will simply shutdown.

If you want to avoid that behaviour, add a `--socket` directive mapping to the required socket (the one that should be managed by the zerg server) and add the `--zerg-fallback` option.

With this setup, if a Zerg server is not available, the Zerg client will continue binding normally to the specified socket(s).

7.4.4 Using Zerg as testers

A good trick you can use, is suspending the main instance with the `SIGTSTP` signal and loading a new version of your app in a Zerg. If the code is not ok you can simply shutdown the Zerg and resume the main instance.

7.4.5 Zerg Pools

Zergpools are special Zerg servers that only serve Zerg clients, nothing more.

You can use them to build high-availability systems that reduce downtime during tests/reloads.

You can run an unlimited number of zerg pools (on several UNIX sockets) and map an unlimited number of sockets to them.

```
[uwsgi]
master = true
zergpool = /tmp/zergpool_1:127.0.0.1:3031,127.0.0.1:3032
zergpool = /tmp/zergpool_2:192.168.173.22:3031,192.168.173.22:3032
```

With a config like this, you will have two zergpools, each serving two sockets.

You can now attach instances to them.

```
uwsgi --zerg /tmp/zergpool_1 --wsgi-file myapp.wsgi --master --processes 8
uwsgi --zerg /tmp/zergpool_2 --rails /var/www/myapp --master --processes 4
```

or you can attach a single instance to multiple Zerg servers.

```
uwsgi --zerg /tmp/zergpool_1 --zerg /tmp/zergpool_2 --wsgi-file myapp.wsgi --master --processes 8
```

7.5 Adding applications dynamically

NOTE: this is not the best approach for hosting multiple applications. You'd better to run a uWSGI instance for each app.

You can start the uWSGI server without configuring an application.

To load a new application you can use these variables in the uwsgi packet:

- `UWSGI_SCRIPT` – pass the name of a WSGI script defining an `application` callable
- or `UWSGI_MODULE` and `UWSGI_CALLABLE` – the module name (importable path) and the name of the callable to invoke from that module

Dynamic apps are officially supported on Cherokee, Nginx, Apache, cgi_dynamic. They are easily addable to the Tomcat and Twisted handlers.

7.5.1 Defining VirtualEnv with dynamic apps

Virtualenvs are based on the `Py_SetPythonHome()` function. This function has effect only if called before `Py_Initialize()` so it can't be used with dynamic apps.

To define a VirtualEnv with DynamicApps, a hack is the only solution.

First you have to tell python to not import the `site` module. This module adds all `site-packages` to `sys.path`. To emulate virtualenvs, we must load the site module only after subinterpreter initialization. Skipping the first `import site`, we can now simply set `sys.prefix` and `sys.exec_prefix` on dynamic app loading and call

```
PyImport_ImportModule("site");
// Some users would want to not disable initial site module loading, so the site module must be reloa
PyImport_ReloadModule(site_module);
```

Now we can set the VirtualEnv dynamically using the `UWSGI_PYHOME` var:

```
location / {
  uwsgi_pass 192.168.173.5:3031;
  include uwsgi_params;
  uwsgi_param UWSGI_SCRIPT mytrac;
  uwsgi_param UWSGI_PYHOME /Users/roberto/uwsgi/VENV2;
}
```

7.6 Scaling SSL connections (uWSGI 1.9)

Distributing SSL servers in a cluster is an hard topic. The biggest problem is sharing SSL sessions between different nodes.

The problem is amplified in non-blocking servers due to OpenSSL's limits in the way sessions are managed.

For example, you cannot share sessions in Memcached servers and access them in a non-blocking way.

A common solution (well, a compromise, maybe) until now has been to use a single SSL terminator balancing requests to multiple non-encrypted backends. This solution kinda works, but obviously it does not scale.

Starting from uWSGI 1.9-dev an implementation (based on the *stud* project) of distributed caching has been added.

7.6.1 Setup 1: using the uWSGI cache for storing SSL sessions

You can configure the SSL subsystem of uWSGI to use the shared cache. The SSL sessions will time out according to the expiry value of the cache item. This way the cache sweeper thread (managed by the master) will destroy sessions in the cache.

```
[uwsgi]
; spawn the master process (it will run the cache sweeper thread)
master = true
; store up to 20k sessions
cache = 20000
; 4k per object is enough for SSL sessions
cache-blocksize = 4096
; force the SSL subsystem to use the uWSGI cache as session storage
ssl-sessions-use-cache = true
; set SSL session timeout (in seconds)
ssl-sessions-timeout = 300
; set the session context string (see later)
https-session-context = foobar
; spawn an HTTPS router
https = 192.168.173.1:8443,foobar.crt,foobar.key
; spawn 8 processes for the HTTPS router (all sharing the same session cache)
http-processes = 8
; add a bunch of uwsgi nodes to relay traffic to
http-to = 192.168.173.10:3031
http-to = 192.168.173.11:3031
http-to = 192.168.173.12:3031
; add stats
stats = 127.0.0.1:5001
```

Now start blasting your HTTPS router and then telnet to port 5001. Under the "cache" object of the JSON output you should see the values "items" and "hits" increasing. The value "miss" is increased every time a session is not found in the cache. It is a good metric of the SSL performance users can expect.

7.6.2 Setup 2: synchronize caches of different HTTPS routers

The objective is to synchronize each new session in each distributed cache. To accomplish that you have to spawn a special thread (`cache-udp-server`) in each instance and list all of the remote servers that should be synchronized.

A pure-TCP load balancer (like HAProxy or uWSGI's Rawrouter) can be used to load balance between the various HTTPS routers.

Here's a possible Rawrouter config.

```
[uwsgi]
master = true
rawrouter = 192.168.173.99:443
rawrouter-to = 192.168.173.1:8443
rawrouter-to = 192.168.173.2:8443
rawrouter-to = 192.168.173.3:8443
```

Now you can configure the first node (the new options are at the end of the .ini config)

```
[uwsgi]
; spawn the master process (it will run the cache sweeper thread)
master = true
; store up to 20k sessions
cache = 20000
; 4k per object is enough for SSL sessions
cache-blocksize = 4096
; force the SSL subsystem to use the uWSGI cache as session storage
ssl-sessions-use-cache = true
; set SSL session timeout (in seconds)
ssl-sessions-timeout = 300
; set the session context string (see later)
https-session-context = foobar
; spawn an HTTPS router
https = 192.168.173.1:8443,foobar.crt,foobar.key
; spawn 8 processes for the HTTPS router (all sharing the same session cache)
http-processes = 8
; add a bunch of uwsgi nodes to relay traffic to
http-to = 192.168.173.10:3031
http-to = 192.168.173.11:3031
http-to = 192.168.173.12:3031
; add stats
stats = 127.0.0.1:5001

; spawn the cache-udp-server
cache-udp-server = 192.168.173.1:7171
; propagate updates to the other nodes
cache-udp-node = 192.168.173.2:7171
cache-udp-node = 192.168.173.3:7171
```

and the other two...

```
[uwsgi]
; spawn the master process (it will run the cache sweeper thread)
master = true
; store up to 20k sessions
cache = 20000
; 4k per object is enough for SSL sessions
cache-blocksize = 4096
; force the SSL subsystem to use the uWSGI cache as session storage
ssl-sessions-use-cache = true
```

```
; set SSL session timeout (in seconds)
ssl-sessions-timeout = 300
; set the session context string (see later)
https-session-context = foobar
; spawn an HTTPS router
https = 192.168.173.1:8443,foobar.crt,foobar.key
; spawn 8 processes for the HTTPS router (all sharing the same session cache)
http-processes = 8
; add a bunch of uwsgi nodes to relay traffic to
http-to = 192.168.173.10:3031
http-to = 192.168.173.11:3031
http-to = 192.168.173.12:3031
; add stats
stats = 127.0.0.1:5001

; spawn the cache-udp-server
cache-udp-server = 192.168.173.2:7171
; propagate updates to the other nodes
cache-udp-node = 192.168.173.1:7171
cache-udp-node = 192.168.173.3:7171

[uwsgi]
; spawn the master process (it will run the cache sweeper thread)
master = true
; store up to 20k sessions
cache = 20000
; 4k per object is enough for SSL sessions
cache-blocksize = 4096
; force the SSL subsystem to use the uWSGI cache as session storage
ssl-sessions-use-cache = true
; set SSL session timeout (in seconds)
ssl-sessions-timeout = 300
; set the session context string (see later)
https-session-context = foobar
; spawn an HTTPS router
https = 192.168.173.1:8443,foobar.crt,foobar.key
; spawn 8 processes for the HTTPS router (all sharing the same session cache)
http-processes = 8
; add a bunch of uwsgi nodes to relay traffic to
http-to = 192.168.173.10:3031
http-to = 192.168.173.11:3031
http-to = 192.168.173.12:3031
; add stats
stats = 127.0.0.1:5001

; spawn the cache-udp-server
cache-udp-server = 192.168.173.3:7171
; propagate updates to the other nodes
cache-udp-node = 192.168.173.1:7171
cache-udp-node = 192.168.173.2:7171
```

Start hammering the Rawrouter (remember to use a client supporting persistent SSL sessions, like your browser) and get cache statistics from the stats server of each HTTPS terminator node. If the count of "hits" is a lot higher than the "miss" value the system is working well and your load is distributed and in awesome hyper high performance mode.

So, what is `https-session-context`, you ask? Basically each SSL session before being used is checked against a fixed string (the session context). If the session does not match that string, it is rejected. By default the session context is initialized to a value built from the HTTP server address. Forcing it to a shared value will avoid a session

created in a node being rejected in another one.

7.6.3 Using named caches

Starting from uWSGI 1.9 you can have multiple caches. This is a setup with 2 nodes using a new generation cache named "ssl".

```
[uwsgi]
; spawn the master process (it will run the cache sweeper thread)
master = true
; store up to 20k sessions
cache2 = name=ssl,items=20000,blocksize=4096,node=127.0.0.1:4242,udp=127.0.0.1:4141
; force the SSL subsystem to use the uWSGI cache as session storage
ssl-sessions-use-cache = ssl
; set sessions timeout (in seconds)
ssl-sessions-timeout = 300
; set the session context string
https-session-context = foobar
; spawn an HTTPS router
https = :8443,foobar.crt,foobar.key
; spawn 8 processes for the HTTPS router (all sharing the same session cache)
http-processes = 8
module = werkzeug.testapp:test_app
; add stats
stats = :5001
```

and the second node...

```
[uwsgi]
; spawn the master process (it will run the cache sweeper thread)
master = true
; store up to 20k sessions
cache2 = name=ssl,items=20000,blocksize=4096,node=127.0.0.1:4141,udp=127.0.0.1:4242
; force the SSL subsystem to use the uWSGI cache as session storage
ssl-sessions-use-cache = ssl
; set session timeout
ssl-sessions-timeout = 300
; set the session context string
https-session-context = foobar
; spawn an HTTPS router
https = :8444,foobar.crt,foobar.key
; spawn 8 processes for the HTTPS router (all sharing the same sessions cache)
http-processes = 8
module = werkzeug.testapp:test_app
; add stats
stats = :5002
```

7.6.4 Notes

If you do not want to manually configure the cache UDP nodes and your network configuration supports it, you can use UDP multicast.

```
[uwsgi]
...
cache-udp-server = 225.1.1.1:7171
cache-udp-node = 225.1.1.1:7171
```

- A new gateway server is in development, named "udprepeater". It will basically forward all of UDP packets it receives to the subscribed back-end nodes. It will allow you to maintain the zero-config style of the subscription system (basically you only need to configure a single cache UDP node pointing to the repeater).
- Currently there is no security between the cache nodes. For some users this may be a huge problem, so a security mode (encrypting the packets) is in development.

CHAPTER 8

Securing uWSGI

8.1 Setting POSIX Capabilities

POSIX capabilities allow fine-grained permissions for processes. In addition to the standard UNIX permission scheme, they define a new set of privileges for system resources. To enable capabilities support (Linux Only) you have to install the `libcap` headers (`libcap-dev` on Debian-based distros) before building uWSGI. As usual your processes will lose practically all of the capabilities after a `setuid` call. The uWSGI `cap` option allows you to define a list of capabilities to maintain through the call.

For example, to allow your unprivileged app to bind on privileged ports and set the system clock, you will use the following options.

```
uwsgi --socket :1000 --uid 5000 --gid 5000 --cap net_bind_service,sys_time
```

All of the processes generated by uWSGI will then inherit this behaviour. If your system supports capabailities not available in the uWSGI list you can simply specify the number of the constant:

```
uwsgi --socket :1000 --uid 5000 --gid 5000 --cap net_bind_service,sys_time,42
```

In addition to `net_bind_service` and `sys_time`, a new capability numbered '42' is added.

8.1.1 Available capabilities

This is the list of available capabilities.

audit_control	CAP_AUDIT_CONTROL
audit_write	CAP_AUDIT_WRITE
chown	CAP_CHOWN
dac_override	CAP_DAC_OVERRIDE
dac_read_search	CAP_DAC_READ_SEARCH
fowner	CAP_FOWNER
fsetid	CAP_FSETID
ipc_lock	CAP_IPC_LOCK
ipc_owner	CAP_IPC_OWNER
kill	CAP_KILL
lease	CAP_LEASE
linux_immutable	CAP_LINUX_IMMUTABLE
mac_admin	CAP_MAC_ADMIN
mac_override	CAP_MAC_OVERRIDE

Continued on next page

Table 8.1 – continued from previous page

mknod	CAP_MKNOD
net_admin	CAP_NET_ADMIN
net_bind_service	CAP_NET_BIND_SERVICE
net_broadcast	CAP_NET_BROADCAST
net_raw	CAP_NET_RAW
setfcap	CAP_SETFCAP
setgid	CAP_SETGID
setpcap	CAP_SETPCAP
setuid	CAP_SETUID
sys_admin	CAP_SYS_ADMIN
sys_boot	CAP_SYS_BOOT
sys_chroot	CAP_SYS_CHROOT
sys_module	CAP_SYS_MODULE
sys_nice	CAP_SYS_NICE
sys_pacct	CAP_SYS_PACCT
sys_ptrace	CAP_SYS_PTRACE
sys_rawio	CAP_SYS_RAWIO
sys_resource	CAP_SYS_RESOURCE
sys_time	CAP_SYS_TIME
sys_tty_config	CAP_SYS_TTY_CONFIG
syslog	CAP_SYSLOG
wake_alarm	CAP_WAKE_ALARM

8.2 Running uWSGI in a Linux CGroup

Linux cgroups are an amazing feature available in recent Linux kernels. They allow you to "jail" your processes in constrained environments with limited CPU, memory, scheduling priority, IO, etc..

Note: uWSGI has to be run as root to use cgroups. `uid` and `gid` are very, very necessary.

8.2.1 Enabling cgroups

First you need to enable cgroup support in your system. Create the /cgroup directory and add this to your /etc/fstab:

```
none /cgroup cgroup cpu,cpuacct,memory
```

Then mount /cgroup and you'll have jails with controlled CPU and memory usage. There are other Cgroup subsystems, but CPU and memory usage are the most useful to constrain.

Let's run uWSGI in a cgroup:

```
./uwsgi -M -p 8 --cgroup /cgroup/jail001 -w simple_app -m --http :9090
```

Cgroups are simple directories. With this command your uWSGI server and its workers are "jailed" in the 'cgroup/jail001' cgroup. If you make a bunch of requests to the server, you will see usage counters – cpuacct.* and memoryfiles.* in the cgroup directory growing. You can also use pre-existing cgroups by specifying a directory that already exists.

8.2.2 A real world example: Scheduling QoS for your customers

Suppose you're hosting apps for 4 customers. Two of them are paying you $100 a month, one is paying $200, and the last is paying $400. To have a good Quality of Service implementation, the $100 apps should get 1/8, or 12.5% of your CPU power, the $200 app should get 1/4 (25%) and the last should get 50%. To implement this, we have to create 4 cgroups, one for each app, and limit their scheduling weights.

```
./uwsgi --uid 1001 --gid 1001 -s /tmp/app1 -w app1 --cgroup /cgroup/app1 --cgroup-opt cpu.shares=125
./uwsgi --uid 1002 --gid 1002 -s /tmp/app2 -w app1 --cgroup /cgroup/app2 --cgroup-opt cpu.shares=125
./uwsgi --uid 1003 --gid 1003 -s /tmp/app3 -w app1 --cgroup /cgroup/app3 --cgroup-opt cpu.shares=250
./uwsgi --uid 1004 --gid 1004 -s /tmp/app4 -w app1 --cgroup /cgroup/app4 --cgroup-opt cpu.shares=500
```

The `cpu.shares` values are simply computed relative to each other, so you can use whatever scheme you like, such as (125, 125, 250, 500) or even (1, 1, 2, 4). With CPU handled, we turn to limiting memory. Let's use the same scheme as before, with a maximum of 2 GB for all apps altogether.

```
./uwsgi --uid 1001 --gid 1001 -s /tmp/app1 -w app1 --cgroup /cgroup/app1 --cgroup-opt cpu.shares=125
./uwsgi --uid 1002 --gid 1002 -s /tmp/app2 -w app1 --cgroup /cgroup/app2 --cgroup-opt cpu.shares=125
./uwsgi --uid 1003 --gid 1003 -s /tmp/app3 -w app1 --cgroup /cgroup/app3 --cgroup-opt cpu.shares=250
./uwsgi --uid 1004 --gid 1004 -s /tmp/app4 -w app1 --cgroup /cgroup/app4 --cgroup-opt cpu.shares=500
```

8.3 Using Linux KSM in uWSGI

Kernel Samepage Merging is a feature of Linux kernels >= 2.6.32 which allows processes to share pages of memory with the same content. This is accomplished by a kernel daemon that periodically performs scans, comparisons, and, if possible, merges of specific memory areas. Born as an enhancement for KVM it can be used for processes that use common data (such as uWSGI processes with language interpreters and standard libraries).

If you are lucky, using KSM may exponentially reduce the memory usage of your uWSGI instances. Especially in massive *Emperor* deployments: enabling KSM for each vassal may result in massive memory savings. KSM in uWSGI was the idea of Giacomo Bagnoli of Asidev s.r.l.. Many thanks to him.

8.3.1 Enabling the KSM daemon

To enable the KSM daemon (`ksmd`), simply set `/sys/kernel/mm/ksm/run` to 1, like so:

```
echo 1 > /sys/kernel/mm/ksm/run
```

Note: Remember to do this on machine startup, as the KSM daemon does not run by default.

Note: KSM is an opt-in feature that has to be explicitly requested by processes, so just enabling KSM will not be a savior for everything on your machine.

8.3.2 Enabling KSM support in uWSGI

If you have compiled uWSGI on a kernel with KSM support, you will be able to use the `ksm` option. This option will instruct uWSGI to register process memory mappings (via `madvice` syscall) after each request or master cycle. If no page mapping has changed from the last scan, no expensive syscalls are used.

8.3.3 Performance impact

Checking for process mappings requires parsing the `/proc/self/maps` file after each request. In some setups this may hurt performance. You can tune the frequency of the uWSGI page scanner by passing an argument to the `ksm` option.

```
# Scan for process mappings every 10 requests (or 10 master cycles)
./uwsgi -s :3031 -M -p 8 -w myapp --ksm=10
```

8.3.4 Check if KSM is working well

The `/sys/kernel/mm/ksm/pages_shared` and `/sys/kernel/mm/ksm/pages_sharing` files contain statistics regarding KSM's efficiency. The higher values, the less memory consumption for your uWSGI instances.

KSM statistics with collectd

A simple Bash script like this is useful for keeping an eye on KSM's efficiency:

```
#!/bin/bash

export LC_ALL=C

if [ -e /sys/kernel/mm/ksm/pages_sharing ]; then
    pages_sharing=`cat /sys/kernel/mm/ksm/pages_sharing`;
    page_size=`getconf PAGESIZE`;
    saved=$(echo "scale=0;$pages_sharing * $page_size"|bc);
    echo "PUTVAL <%= cn %>/ksm/gauge-saved interval=60 N:$saved"
fi
```

In your collectd configuration, add something like this:

```
LoadPlugin exec
<Plugin exec>
   Exec "nobody" "/usr/local/bin/ksm_stats.sh"
</Plugin>
```

8.4 Jailing your apps using Linux Namespaces

If you have a recent Linux kernel (>2.6.26) you can use its support for namespaces.

8.4.1 What are namespaces?

They are an elegant (more elegant than most of the jailing systems you might find in other operating systems) way to "detach" your processes from a specific layer of the kernel and assign them to a new one.

The 'chroot' system available on UNIX/Posix systems is a primal form of namespaces: a process sees a completely new file system root and has no access to the original one.

Linux extends this concept to the other OS layers (PIDs, users, IPC, networking etc.), so a specific process can live in a "virtual OS" with a new group of pids, a new set of users, a completely unshared IPC system (semaphores, shared memory etc.), a dedicated network interface and its own hostname.

uWSGI got full namespaces support in 1.9/2.0 development cycle.

8.4.2 clone() vs unshare()

To place the current process in a new namespace you have two syscall: the venerable clone(), that will create a new process in the specified namespaces and the new unshare() that changes namespaces of the currently running process.

clone() can be used by the Emperor to directly spawn vassals in new namespaces:

```
[uwsgi]
emperor = /etc/uwsgi/vassals
emperor-use-clone = fs,net,ipc,uts,pid
```

will run each vassal with a dedicated filesystems, networking, sysv ipc and uts view

while

```
[uwsgi]
unshare = ipc,uts
...
```

will run the instance in the specified namespaces.

Some namespace subsystem requires additional steps (see below)

8.4.3 Supported namespaces

`fs` -> CLONE_NEWNS, filesystems

`ipc` -> CLONE_NEWIPC, sysv ipc

`pid` -> CLONE_NEWPID, when used with unshare() requires an additional fork(), use one of the –refork-* options

`uts` -> CLONE_NEWUTS, hostname

`net` -> CLONE_NEWNET, new networking, UNIX sockets from different namespaces are still usable, they are a good way for inter-namespaces communications

`user` -> CLONE_NEWUSER, still complex to manage (and has differences in behaviours between kernel versions) use with caution

8.4.4 setns()

In addition to creating new namespaces for a process you can attach to already running ones using the setns() call.

Each process exposes its namespaces via the /proc/self/ns directory. The setns() syscall uses the filedescriptors obtained from the files in that directory to attach to namespaces.

As we have already seen, unix sockets are a good way to communicate between namespaces, the uWSGI setns() features works by createing a unix socket that receives requests from processes wanting to join its namespace. As UNIX sockets allow file descriptors passing, the “client” only need to call setns() on them.

`setns-socket <addr>` exposes /proc/self/ns on the specified unix socket address

`setns <addr>` connect to the specified unix socket address, get the filedescriptors and use setns() on them

`setns-preopen` if enabled the /proc/self/ns files are opened on startup (before privileges drop) and cached. This is useful for avoiding running the main instance as root.

`setns-socket-skip <name>` some file in /proc/self/ns can create problems (mostly the ‘user’ one). You can skip them specifying the name. (you can specify this option multiple times)

8.4.5 pivot_root

This option allows you to change the rootfs of your currently running instance.

It is better than chroot as it allows you to access the old filesystem tree before (manually) unmounting it.

It is a bit complex to master correctly as it requires a couple of assumptions:

```
pivot_root <new> <old>
```

<new> is the directory to mount as the new rootfs and <old> is where to access the old tree.

<new> must be a mounted filesystem, and <old> must be under this filesystem.

A common pattern is:

```
[uwsgi]
unshare = fs
hook-post-jail = mount:none /distros/precise /ns bind
pivot_root = /ns /ns/.old_root
...
```

remember to create /ns and /distro/precise/.old_root

When you have created the new filesysten layout you can umount /.old_root recursively:

```
[uwsgi]
unshare = fs
hook-post-jail = mount:none /distros/precise /ns bind
pivot_root = /ns /ns/.old_root
; bind mount some useful fs like /dev and /proc
hook-as-root = mount:proc none /proc nodev hidepid=2
hook-as-root = mount:none /.old_root/dev /dev bind
hook-as-root = mount:none /.old_root/dev/pts /dev/pts bind
; umount the old tree
hook-as-root = umount:/.old_root rec,detach
```

8.4.6 Why not lxc ?

Lxc is a project allowing you to build full subsystems using linux namespaces. You may ask why "reinventing the wheel" while lxc implements fully "virtualized" system. Apple and oranges.

Lxc objective is giving users the view of a virtual server, uWSGI namespaces support is lower level, you can use it to detach single components (for example you may only want to unshare ipc) to increase security and isolation.

Not all the scenario requires a full system-like view (and in lot of case is suboptimal, while in other is the best approach), try to see namespaces as a way to increase security and isolation, when you need/can isolate a component do it with clone/unshare. When you want to give users a full system-like access go with lxc.

8.5 The old way: the –namespace option

Before 1.9/2.0 a full featured system-like namespace support was added. It works as a chroot() on steroids.

It should be moved as an external plugin pretty soon, but will be always part of the main distribution, as it is used by lot of people for its simplicity.

You basically need to set a root filesystem and an hostname to start your instance in a new namespace:

Let's start by creating a new root filesystem for our jail. You'll need `debootstrap`. We're placing our rootfs in `/ns/001`, and then create a 'uwsgi' user that will run the uWSGI server. We will use the chroot command to 'adduser' in the new rootfs, and we will install the Flask package, required by uwsgicc.

(All this needs to be executed as root)

```
mkdir -p /ns/001
debootstrap maverick /ns/001
chroot /ns/001
# in the chroot jail now
adduser uwsgi
apt-get install mercurial python-flask
su - uwsgi
# as uwsgi now
git clone https://github.com/unbit/uwsgicc.git .
exit # out of uwsgi
exit # out of the jail
```

Now on your real system run

```
uwsgi --socket 127.0.0.1:3031 --chdir /home/uwsgi/uwsgi --uid uwsgi --gid uwsgi --module uwsgicc --ma
```

If all goes well, uWSGI will set `/ns/001` as the new root filesystem, assign `mybeautifulhostname` as the hostname and hide the PIDs and IPC of the host system.

The first thing you should note is the uWSGI master becoming the pid 1 (the "init" process). All processes generated by the uWSGI stack will be reparented to it if something goes wrong. If the master dies, all jailed processes die.

Now point your webbrowser to your webserver and you should see the uWSGI Control Center interface.

Pay attention to the information area. The nodename (used by cluster subsystem) matches the real hostname as it does not make sense to have multiple jail in the same cluster group. In the hostname field instead you will see the hostname you have set.

Another important thing is that you can see all the jail processes from your real system (they will have a different set of PIDs), so if you want to take control of the jail you can easily do it.

Note: A good way to limit hardware usage of jails is to combine them with the cgroups subsystem.

See also:

Running uWSGI in a Linux CGroup

8.5.1 Reloading uWSGI

When running jailed, uWSGI uses another system for reloading: it'll simply tell workers to bugger off and then exit. The parent process living outside the namespace will see this and respawn the stack in a new jail.

8.5.2 How secure is this sort of jailing?

Hard to say! All software tends to be secure until a hole is found.

8.5.3 Additional filesystems

When app is jailed to namespace it only has access to its virtual jail root filesystem. If there is any other filesystem mounted inside the jail directory, it won't be accessible, unless you use `namespace-keep-mount`.

```
# app1 jail is located here
namespace = /apps/app1

# nfs share mounted on the host side
namespace-keep-mount = /apps/app1/nfs
```

This will bind /apps/app1/nfs to jail, so that jailed app can access it under /nfs directory

```
# app1 jail is located here
namespace = /apps/app1

# nfs share mounted on the host side
namespace-keep-mount = /mnt/nfs1:/nfs
```

If the filesystem that we want to bind is mounted in path not contained inside our jail, than we can use "<source>:<dest>" syntax for –namespace-keep-mount. In this case the /mnt/nfs1 will be binded to /nfs directory inside the jail.

8.6 FreeBSD Jails

uWSGI 1.9.16 introduced native FreeBSD jails support.

FreeBSD jails can be seen as new-generation chroot() with fine-grained tuning of what this "jail" can see.

They are very similar to Linux namespaces even if a bit higher-level (from the API point of view).

Jails are available since FreeBSD 4

8.6.1 Why managing jails with uWSGI ?

Generally jails are managed using the system tool "jail" and its utilities.

Til now running uWSGI in FreeBSD jails was pretty common, but for really massive setups (read: hosting business) where an Emperor (for example) manages hundreds of unrelated uWSGI instances, the setup could be really overkill.

Managing jails directly in uWSGI config files highly reduce sysadmin costs and helps having a better organization of the whole infrastructure.

8.6.2 Old-style jails (FreeBSD < 8)

FreeBSD exposes two main api for managing jails. The old (and easier) one is based on the jail() function.

It is available since FreeBSD 4 and allows you to set the rootfs, the hostname and one ore more ipv4/ipv6 addresses

Two options are needed for running a uWSGI instance in a jail: –jail and –jail-ip4/–jail-ip6 (effectively they are 3 if you use IPv6)

`--jail <rootfs> [hostname] [jailname]`

`--jail-ip4 <address>` (can be specified multiple times)

`--jail-ip6 <address>` (can be specified multiple times)

Showing how to create the rootfs for your jail is not the objective of this document, but personally i hate rebuilding from sources, so generally i simply explode the base.tgz file from an official repository and chroot() to it to make the fine tuning.

An important thing you have to remember is that the ip addresses you attach to a jail must be available in the system (as aliases). As always we tend to abuse uWSGI facilities. In our case the –exec-pre-jail hook will do the trick

```
[uwsgi]
; create the jail with /jails/001 as rootfs and 'foobar' as hostname
jail = /jails/001 foobar
; create the alias on 'em0'
exec-pre-jail = ifconfig em0 192.168.0.40 alias
; attach the alias to the jail
jail-ip4 = 192.168.0.40

; bind the http-socket (we are now in the jail)
http-socket = 192.168.0.40:8080

; load the application (remember we are in the jail)
wsgi-file = myapp.wsgi

; drop privileges
uid = kratos
gid = kratos

; common options
master = true
processes = 2
```

8.6.3 New style jails (FreeBSD >= 8)

FreeBSD 8 introdiced a new advanced api for managing jails. Based on the jail_set() syscall, libjail exposes dozens of features and allows fine-tuning of your jails. To use the new api you need the –jail2 option (aliased as –libjail)

```
--jail2 <key>[=value]
```

Each –jail2 option maps 1:1 with a jail attribute so you can basically tune everything !

```
[uwsgi]
; create the jail with /jails/001 as rootfs
jail2 = path=/jails/001
; set hostname to 'foobar'
jail2 = host.hostname=foobar
; create the alias on 'em0'
exec-pre-jail = ifconfig em0 192.168.0.40 alias
; attach the alias to the jail
jail2 = ip4.addr=192.168.0.40

; bind the http-socket (we are now in the jail)
http-socket = 192.168.0.40:8080

; load the application (remember we are in the jail)
wsgi-file = myapp.wsgi

; drop privileges
uid = kratos
gid = kratos

; common options
master = true
processes = 2
```

Note for FreeBSD >= 8.4 but < 9.0

uWSGI uses ipc semaphores on FreeBSD < 9 (newer FreeBSD releases have POSIX semaphores support).

Since FreeBSD 8.4 you need to explicitely allows sysvipc in jails. So be sure to have

```
[uwsgi]
...
jail2 = allow.sysvipc=1
...
```

8.6.4 DevFS

The DevFS virtual filesystem manages the /dev directory on FreeBSD.

The /dev filesystem is not mounted in the jail, but you can need it for literally hundreds of reasons.

Two main approaches are available: mounting it in the /dev/ directory of the roots before creating the jail, or allowing the jail to mount it

```
[uwsgi]
; avoid re-mounting the file system every time
if-not-exists = /jails/001/dev/zero
  exec-pre-jail = mount -t devfs devfs /jails/001/dev
endif =
; create the jail with /jails/001 as rootfs
jail2 = path=/jails/001
; set hostname to 'foobar'
jail2 = host.hostname=foobar
; create the alias on 'em0'
exec-pre-jail = ifconfig em0 192.168.0.40 alias
; attach the alias to the jail
jail2 = ip4.addr=192.168.0.40

; bind the http-socket (we are now in the jail)
http-socket = 192.168.0.40:8080

; load the application (remember we are in the jail)
wsgi-file = myapp.wsgi

; drop privileges
uid = kratos
gid = kratos

; common options
master = true
processes = 2
```

or (allow the jail itself to mount it)

```
[uwsgi]
; create the jail with /jails/001 as rootfs
jail2 = path=/jails/001
; set hostname to 'foobar'
jail2 = host.hostname=foobar
; create the alias on 'em0'
exec-pre-jail = ifconfig em0 192.168.0.40 alias
; attach the alias to the jail
jail2 = ip4.addr=192.168.0.40
```

```
; allows mount of devfs in the jail
jail2 = enforce_statfs=1
jail2 = allow.mount
jail2 = allow.mount.devfs
; ... and mount it
if-not-exists = /dev/zero
  exec-post-jail = mount -t devfs devfs /dev
endif =

; bind the http-socket (we are now in the jail)
http-socket = 192.168.0.40:8080

; load the application (remember we are in the jail)
wsgi-file = myapp.wsgi

; drop privileges
uid = kratos
gid = kratos

; common options
master = true
processes = 2
```

8.6.5 Reloading

Reloading (or binary patching) is a bit annoying to manage as uWSGI need to re-exec itself, so you need a copy of the binary, plugins and the config file in your jail (unless you can sacrifice graceful reload and simply delegate the Emperor to respawn the instance)

Another approach is (like with devfs) mounting the directory with the uwsgi binary (and the eventual plugins) in the jail itself and instruct uWSGI to use this new path with –binary-path

8.6.6 The jidfile

Each jail can be referenced by a unique name (optional) or its "jid". This is similar to a "pid", as you can use it to send commands (and updates) to an already running jail. The –jidfile <file> option allows you to store the jid in a file for use with external applications.

8.6.7 Attaching to a jail

You can attach uWSGI instances to already running jails (they can be standard persistent jail too) using –jail-attach <id>

The id argument can be a jid or the name of the jail.

This feature requires FreeBSD 8

8.6.8 Debian/kFreeBSD

This is an official Debian project aiming at building an os with FreeBSD kernel and common Debian userspace.

It works really well, and it has support for jails too.

Let's create a jail with debootstrap

```
debootstrap wheezy /jails/wheezy
```

add a network alias

```
ifconfig em0 192.168.173.105 netmask 255.255.255.0 alias
```

(change em0 with your network interface name)

and run it

```
uwsgi --http-socket 192.168.173.105:8080 --jail /jails/wheezy -jail-ip4 192.168.173.105
```

8.6.9 Jails with Forkpty Router

You can easily attach to FreeBSD jails with *The Forkpty Router*

Just remember to have /dev (well, /dev/ptmx) mounted in your jail to allow the forkpty() call

Learn how to deal with devfs_ruleset to increase security of your devfs

8.6.10 Notes

A jail is destroyed when the last process running in it dies

By default everything mounted under the rootfs (before entering the jail) will be seen by the jail it self (we have seen it before when dealing with devfs)

8.7 The Forkpty Router

Dealing with containers is now a common deployment pattern. One of the most annoying tasks when dealing with jails/namespaces is 'attaching' to already running instances.

The forkpty router aims at simplifyng the process giving a pseudoterminal server to your uWSGI instances.

A client connect to the socket exposed by the forkpty router and get a new pseudoterminal connected to a process (generally a shell, but can be whatever you want)

8.7.1 uwsgi mode VS raw mode

Clients connecting to the forkpty router can use two protocols for data exchange: uwsgi and raw mode.

The raw mode simply maps the socket to the pty, for such a reason you will not be able to resize your terminal or send specific signals. The advantage of this mode is in performance: no overhead for each char.

The uwsgi mode encapsulates every instruction (stdin, signals, window changes) in a uwsgi packet. This is very similar to how ssh works, so if you plan to use the forkpty router for shell sessions the uwsgi mode is the best choice (in terms of user experience).

The overhead of the uwsgi protocol (worst case) is 5 bytes for each stdin event (single char)

8.7.2 Running the forkpty router

The plugin is not builtin by default, so you have to compile it:

```
python uwsgiconfig.py --plugin plugins/forkptyrouter
```

generally compiling the pty plugin is required too (for client access)

```
python uwsgiconfig.py --plugin plugins/pty
```

you can build al in one shot with:

```
UWSGI_EMBED_PLUGINS=pty,forkptyrouter make
```

Now you can run the forkptyrouter as a standard gateway (we use UNIX socket as we want a communication channel with jails, and we unshare the uts namespace to give a new hostname)

```
[uwsgi]
master = true
unshare = uts
exec-as-root = hostname iaminajail
uid = kratos
gid = kratos
forkpty-router = /tmp/fpty.socket
```

and connect with the pty client:

```
uwsgi --pty-connect /tmp/fpty.socket
```

now you have a shell (/bin/sh by default) in the uWSGI instance. Running `hostname` will give you 'iaminajail'

The previous example uses raw mode, if you resize the client terminal you will se no updates.

To use the 'uwsgi' mode add a 'u':

```
[uwsgi]
master = true
unshare = uts
exec-as-root = hostname iaminajail
uid = kratos
gid = kratos
forkpty-urouter = /tmp/fpty.socket

uwsgi --pty-uconnect /tmp/fpty.socket
```

a single instance can expose both protocols on different sockets

```
[uwsgi]
master = true
unshare = uts
exec-as-root = hostname iaminajail
uid = kratos
gid = kratos
forkpty-router = /tmp/raw.socket
forkpty-urouter = /tmp/uwsgi.socket
```

8.7.3 Changing the default command

By default the forkpty router run /bin/sh on new connections.

You can change the command using the –forkptyrouter-command

```
[uwsgi]
master = true
unshare = uts
exec-as-root = hostname iaminajail
uid = kratos
gid = kratos
forkpty-router = /tmp/raw.socket
forkpty-urouter = /tmp/uwsgi.socket
forkptyrouter-command= /bin/zsh
```

8.8 The TunTap Router

The TunTap router is an ad-hoc solution for giving network connectivity to Linux processes running in a dedicated network namespace (well obviously it has other uses, but very probably this is the most interesting one, and the one for which it was developed)

The TunTap router is not compiled in by default.

For having it in one shot:

```
UWSGI_EMBED_PLUGINS=tuntap make
```

(yes the plugin is named only 'tuntap' as effectively it exposes various tuntap devices features)

The best way to use it is binding it to a unix socket, allowing processes in new namespaces to reach it (generally unix sockets are the best communication channel for linux namespaces).

8.8.1 The first config

We want our vassals to live in the 192.168.0.0/24 network, with 192.168.0.1 as default gateway.

The default gateway (read: the tuntap router) is managed by the Emperor itself

```
[uwsgi]
; create the tun device 'emperor0' and bind it to a unix socket
tuntap-router = emperor0 /tmp/tuntap.socket
; give it an ip address
exec-as-root = ifconfig emperor0 192.168.0.1 netmask 255.255.255.0 up
; setup nat
exec-as-root = iptables -t nat -F
exec-as-root = iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
; enable linux ip forwarding
exec-as-root = echo 1 >/proc/sys/net/ipv4/ip_forward
; force vassals to be created in a new network namespace
emperor-use-clone = net
emperor = /etc/vassals
```

The vassals spawned by this Emperor will born without network connectivity.

To give them access to the public network we create a new tun device (it will exist only in the vassal network namespace) instructing it to route traffic to the Emperor tuntap unix socket:

```
[uwsgi]
; create uwsgi0 tun interface and force it to connect to the Emperor exposed unix socket
tuntap-device = uwsgi0 /tmp/tuntap.socket
```

```
; bring up loopback
exec-as-root = ifconfig lo up
; bring up interface uwsgi0
exec-as-root = ifconfig uwsgi0 192.168.0.2 netmask 255.255.255.0 up
; and set the default gateway
exec-as-root = route add default gw 192.168.0.1
; classic options
uid = customer001
gid = customer001
socket = /var/www/foobar.socket
psgi-file = foobar.pl
...
```

8.8.2 The embedded firewall

The TunTap router includes a very simple firewall for governing vassal's traffic

Firewalling is based on 2 chains (in and out), and each rule is formed by 3 parameters: <action> <src> <dst>

The firewall is applied to traffic from the clients to the tuntap device (out) and the opposite (in)

The first matching rule stops the chain, if no rule applies, the policy is "allow"

the following rules allows access from vassals to the internet, but block vassals intercommunication

```
[uwsgi]
tuntap-router = emperor0 /tmp/tuntap.socket

tuntap-router-firewall-out = allow 192.168.0.0/24 192.168.0.1
tuntap-router-firewall-out = deny 192.168.0.0/24 192.168.0.0/24
tuntap-router-firewall-out = allow 192.168.0.0/24 0.0.0.0
tuntap-router-firewall-out = deny
tuntap-router-firewall-in = allow 192.168.0.1 192.168.0.0/24
tuntap-router-firewall-in = deny 192.168.0.0/24 192.168.0.0/24
tuntap-router-firewall-in = allow 0.0.0.0 192.168.0.0/24
tuntap-router-firewall-in = deny

exec-as-root = ifconfig emperor0 192.168.0.1 netmask 255.255.255.0 up
; setup nat
exec-as-root = iptables -t nat -F
exec-as-root = iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
; enable linux ip forwarding
exec-as-root = echo 1 >/proc/sys/net/ipv4/ip_forward
; force vassals to be created in a new network namespace
emperor-use-clone = net
emperor = /etc/vassals
```

8.8.3 Security

The "switching" part of the TunTap router (read: mapping ip addresses to vassals) is pretty simple: the first packet received from a vassal by the TunTap router register the vassal for that ip address. A good approach (from a security point of view) is sending a ping packet soon after network setup in the vassal:

```
[uwsgi]
; create uwsgi0 tun interface and force it to connect to the Emperor exposed unix socket
tuntap-device = uwsgi0 /tmp/tuntap.socket
; bring up loopback
```

```
exec-as-root = ifconfig lo up
; bring up interface uwsgi0
exec-as-root = ifconfig uwsgi0 192.168.0.2 netmask 255.255.255.0 up
; and set the default gateway
exec-as-root = route add default gw 192.168.0.1

; ping something to register
exec-as-root = ping -c 1 192.168.0.1

; classic options
...
```

after a vassal/ip pair is registered, only that combo will be valid (so other vassals will not be able to use that address until the one holding it dies)

8.8.4 The Future

This is becoming a very important part of the unbit.it networking stack. We are currently working on:

- dynamic firewall rules (luajit resulted a great tool for writing fast networking rules)
- federation/proxy of tuntap router (the tuntaprouter can multiplex vassals networking over a tcp connection to an external tuntap router [that is why you can bind a tuntap router to a tcp address])
- authentication of vassals (maybe the old UNIX ancillary credentials could be enough)
- a stats server for network statistics (rx/tx/errors)
- a bandwidth shaper based on the blastbeat project

CHAPTER 9

Keeping an eye on your apps

9.1 Monitoring uWSGI with Nagios

The official uWSGI distribution includes a plugin adding Nagios-friendly output.

To monitor, and eventually get warning messages, via Nagios, launch the following command, where `node` is the socket (UNIX or TCP) to monitor.

```
uwsgi --socket <node> --nagios
```

9.1.1 Setting warning messages

You can set a warning message directly from your app with the `uwsgi.set_warning_message()` function. All ping responses (used by Nagios too) will report this message.

9.2 The embedded SNMP server

The uWSGI server embeds a tiny SNMP server that you can use to integrate your web apps with your monitoring infrastructure.

To enable SNMP support, you must run the uWSGI UDP server and choose a SNMP community string (which is the rudimentary authentication system used by SNMP).

```
./uwsgi -s :3031 -w staticfilesnmp --udp 192.168.0.1:2222 --snmp --snmp-community foo
# or the following. Using the SNMP option to pass the UDP address is a lot more elegant. ;)
./uwsgi -s :3031 -w myapp --master --processes 4 --snmp=192.168.0.1:2222 --snmp-community foo
```

This will run the uWSGI server on TCP port 3031 and UDP port 2222 with SNMP enabled with "foo" as the community string.

Please note that the SNMP server is started in the master process after dropping the privileges. If you want it to listen on a privileged port, you can either use *Capabilities* on Linux, or use the `master-as-root` option to run the master process as root. The `staticfilesnmp.py` file is included in the distribution and is a simple app that exports a counter via SNMP.

The uWSGI SNMP server exports 2 group of information:

- General information is managed by the uWSGI server itself. The base OID to access uWSGI SNMP information is `1.3.6.1.4.1.35156.17`

(`iso.org.dod.internet.private.enterprise.unbit.uwsgi`). General options are mapped to `1.3.6.1.4.1.35156.17.1.x`.

- Custom information is managed by the apps and accessed via `1.3.6.1.4.1.35156.17.2.x`

So, to get the number of requests managed by the uWSGI server, you could do

```
snmpget -v2c -c foo 192.168.0.1:2222 1.3.6.1.4.1.35156.17.1.1 # 1.1 corresponds to ''general.requests
```

9.2.1 Exporting custom values

To manage custom values from your app you have these Python functions,

- `uwsgi.snmp_set_counter32()`
- `uwsgi.snmp_set_counter64()`
- `uwsgi.snmp_set_gauge()`
- `uwsgi.snmp_incr_counter32()`
- `uwsgi.snmp_incr_counter64()`
- `uwsgi.snmp_incr_gauge()`
- `uwsgi.snmp_decr_counter32()`
- `uwsgi.snmp_decr_counter64()`
- `uwsgi.snmp_decr_gauge()`

So if you wanted to export the number of users currently logged in (this is a gauge as it can lower) as custom OID 40, you'd call

```
users_logged_in = random.randint(0, 1024) # a more predictable source of information would be better
uwsgi.snmp_set_gauge(40, users_logged_in)
```

and to look it up,

```
snmpget -v2c -c foo 192.168.0.1:2222 1.3.6.1.4.1.35156.17.2.40
```

The system snmp daemon (net-snmp) can be configured to proxy SNMP requests to uwsgi. This allows you to run the system daemon and uwsgi at the same time, and runs all SNMP requests through the system daemon first. To configure the system snmp daemon (net-snmp) to proxy connections to uwsgi, add these lines to the bottom of /etc/snmp/snmpd.conf and restart the daemon:

```
proxy -v 2c -c foo 127.0.0.1:2222 .1.3.6.1.4.1.35156.17
view    systemview    included   .1.3.6.1.4.1.35156.17
```

Replace 'foo' and '2222' with the community and port configured in uwsgi.

9.3 Pushing statistics (from 1.4)

IMPORTANT: the Metrics subsystem offers a better introduction to the following concepts. See *The Metrics subsystem*

Starting from uWSGI 1.4 you can push statistics (the same JSON blob you get with the *The uWSGI Stats Server*) via various systems (called stats pushers).

Statistics are pushed at regular intervals (default 3 seconds).

9.3.1 The 'file' stats pusher

By default the 'file' stats pusher is available up to 1.9.18. Starting from 1.9.19 is available as a plugin (stats_pusher_file).

It allows you to save json chunks to a file (open in appended mode)

```
[uwsgi]
socket = :3031
module = foobar
master = true
stats-push = file:path=/tmp/foobar,freq=10
```

this config will append JSON to the /tmp/foobar file every 10 seconds

9.3.2 The 'mongodb' stats pusher

This is the first developed stats pusher plugin, allowing you to store JSON data directly on a mongodb collection

```
[uwsgi]
plugins = stats_pusher_mongodb
socket = :3031
module = foobar
master = true
stats-push = mongodb:addr=127.0.0.1:5151,collection=uwsgi.mystats,freq=4
```

This config will insert JSON data to the collection uwsgi.mystats on the mongodb server 127.0.0.1:5151 every 4 seconds.

To build the plugin you need mongodb development headers (mongodb-dev on Debian/Ubuntu)

```
python uwsgiconfig.py --plugin plugins/stats_pusher_mongodb
```

will do the trick

9.3.3 Notes

You can configure all of the stats pusher you need, just specify multiple stats-push options

```
[uwsgi]
plugins = stats_pusher_mongodb
socket = :3031
module = foobar
master = true
stats-push = mongodb:addr=127.0.0.1:5151,collection=uwsgi.mystats,freq=4
stats-push = mongodb:addr=127.0.0.1:5152,collection=uwsgi.mystats,freq=4
stats-push = mongodb:addr=127.0.0.1:5153,collection=uwsgi.mystats,freq=4
stats-push = mongodb:addr=127.0.0.1:5154,collection=uwsgi.mystats,freq=4
```

9.4 Integration with Graphite/Carbon

Graphite is a kick-ass realtime graphing application built on top of three components:

- Whisper – a data storage system
- Carbon – a server for receiving data

- Python web application for graph rendering and management.

The uWSGI Carbon plugin allows you to send uWSGI's internal statistics to one or more Carbon servers. It is compiled in by default as of uWSGI 1.0, though it can also be built as a plugin.

9.4.1 Quickstart

For the sake of illustration, let's say your Carbon server is listening on `127.0.0.1:2003` and your uWSGI instance is on the machine `debian32`, listening on `127.0.0.1:3031` with 4 processes. By adding the `--carbon` option to your uWSGI instance you'll instruct it to send its statistics to the Carbon server periodically. The default period is 60 seconds.

```
uwsgi --socket 127.0.0.1:3031 --carbon 127.0.0.1:2003 --processes 4
```

Metrics are named like `uwsgi.<hostname>.<id>.requests` and `uwsgi.<hostname>.<id>.worker<n>.requests`, where:

- `hostname` – machine's hostname
- `id` – name of the first uWSGI socket (with dots replaced by underscores)
- `n` – number of the worker processes (1-based).

Examples of names of Carbon metrics generated by uWSGI:

- `uwsgi.debian32.127_0_0_1:3031.requests` (`uwsgi.<hostname>.<id>.requests`)
- `uwsgi.debian32.127_0_0_1:3031.worker1.requests` (`uwsgi.<hostname>.<id>.worker<n>.requests`)
- `uwsgi.debian32.127_0_0_1:3031.worker2.requests` (`uwsgi.<hostname>.<id>.worker<n>.requests`)
- `uwsgi.debian32.127_0_0_1:3031.worker3.requests` (`uwsgi.<hostname>.<id>.worker<n>.requests`)
- `uwsgi.debian32.127_0_0_1:3031.worker4.requests` (`uwsgi.<hostname>.<id>.worker<n>.requests`)

See also:

Setting up Graphite on Ubuntu using the Metrics subsystem

9.5 The uWSGI Stats Server

In addition to *SNMP*, uWSGI also supports a Stats Server mechanism which exports the uWSGI state as a JSON object to a socket.

Simply use the `stats` option followed by a valid socket address.

```
--stats 127.0.0.1:1717
--stats /tmp/statsock
--stats :5050
--stats @foobar
```

If a client connects to the specified socket it will get a JSON object containing uWSGI internal statistics before the connection ends.

```
uwsgi --socket :3031 --stats :1717 --module welcome --master --processes 8
```

then

```
nc 127.0.0.1 1717
# or for convenience...
uwsgi --connect-and-read 127.0.0.1:1717
```

will return something like this:


```
{
  "workers": [{
    "id": 1,
    "pid": 31759,
    "requests": 0,
    "exceptions": 0,
    "status": "idle",
    "rss": 0,
    "vsz": 0,
    "running_time": 0,
    "last_spawn": 1317235041,
    "respawn_count": 1,
    "tx": 0,
    "avg_rt": 0,
    "apps": [{
      "id": 0,
      "modifier1": 0,
      "mountpoint": "",
      "requests": 0,
      "exceptions": 0,
      "chdir": ""
    }]
  }, {
    "id": 2,
    "pid": 31760,
    "requests": 0,
    "exceptions": 0,
    "status": "idle",
    "rss": 0,
    "vsz": 0,
    "running_time": 0,
    "last_spawn": 1317235041,
    "respawn_count": 1,
    "tx": 0,
    "avg_rt": 0,
    "apps": [{
      "id": 0,
      "modifier1": 0,
      "mountpoint": "",
      "requests": 0,
      "exceptions": 0,
      "chdir": ""
    }]
  }, {
    "id": 3,
    "pid": 31761,
    "requests": 0,
    "exceptions": 0,
    "status": "idle",
    "rss": 0,
    "vsz": 0,
    "running_time": 0,
    "last_spawn": 1317235041,
    "respawn_count": 1,
    "tx": 0,
    "avg_rt": 0,
    "apps": [{
```

```
      "id": 0,
      "modifier1": 0,
      "mountpoint": "",
      "requests": 0,
      "exceptions": 0,
      "chdir": ""
    }]
  }, {
    "id": 4,
    "pid": 31762,
    "requests": 0,
    "exceptions": 0,
    "status": "idle",
    "rss": 0,
    "vsz": 0,
    "running_time": 0,
    "last_spawn": 1317235041,
    "respawn_count": 1,
    "tx": 0,
    "avg_rt": 0,
    "apps": [{
      "id": 0,
      "modifier1": 0,
      "mountpoint": "",
      "requests": 0,
      "exceptions": 0,
      "chdir": ""
    }]
  }, {
    "id": 5,
    "pid": 31763,
    "requests": 0,
    "exceptions": 0,
    "status": "idle",
    "rss": 0,
    "vsz": 0,
    "running_time": 0,
    "last_spawn": 1317235041,
    "respawn_count": 1,
    "tx": 0,
    "avg_rt": 0,
    "apps": [{
      "id": 0,
      "modifier1": 0,
      "mountpoint": "",
      "requests": 0,
      "exceptions": 0,
      "chdir": ""
    }]
  }, {
    "id": 6,
    "pid": 31764,
    "requests": 0,
    "exceptions": 0,
    "status": "idle",
    "rss": 0,
    "vsz": 0,
    "running_time": 0,
```

```
    "last_spawn": 1317235041,
    "respawn_count": 1,
    "tx": 0,
    "avg_rt": 0,
    "apps": [{
      "id": 0,
      "modifier1": 0,
      "mountpoint": "",
      "requests": 0,
      "exceptions": 0,
      "chdir": ""
    }]
  }, {
    "id": 7,
    "pid": 31765,
    "requests": 0,
    "exceptions": 0,
    "status": "idle",
    "rss": 0,
    "vsz": 0,
    "running_time": 0,
    "last_spawn": 1317235041,
    "respawn_count": 1,
    "tx": 0,
    "avg_rt": 0,
    "apps": [{
      "id": 0,
      "modifier1": 0,
      "mountpoint": "",
      "requests": 0,
      "exceptions": 0,
      "chdir": ""
    }]
  }, {
    "id": 8,
    "pid": 31766,
    "requests": 0,
    "exceptions": 0,
    "status": "idle",
    "rss": 0,
    "vsz": 0,
    "running_time": 0,
    "last_spawn": 1317235041,
    "respawn_count": 1,
    "tx": 0,
    "avg_rt": 0,
    "apps": [{
      "id": 0,
      "modifier1": 0,
      "mountpoint": "",
      "requests": 0,
      "exceptions": 0,
      "chdir": ""
    }]
  }]
}
```

9.5.1 uwsgitop

`uwsgitop` is a top-like command that uses the stats server. It is available on PyPI, so use `easy_install` or `pip` to install it (package name `uwsgitop`, naturally).

The sources are available on Github. https://github.com/unbit/uwsgitop

9.6 The Metrics subsystem

(available from 1.9.19)

The uWSGI metrics subsystem allows you to manage "numbers" from your app.

While the caching subsystem got math capabilities during 1.9 development cycle, the metrics subsystem is optimized by design for storing numbers and applying functions over them. So compared to the caching subsystem is way faster and requires a fraction of the memory.

When enabled, the metric subsystem configures a vast amount of metrics (like requests per-core, memory usage...) but, in addition to this, you can configure your own metrics (for example you can account the number users or the hit of a particular url, as well as the memory consumption of your app or the whole server)

To enable the metrics subsystem just add `--enable-metrics` to your options, or configure a stats pusher (see below).

The metrics subsystem is totally thread-safe

By default uWSGI creates a lot of metrics (and mores are planned) so before adding your own, be sure uWSGI does not already expose the one you need.

9.6.1 Metric names and oids

Each metric must have a name (containing only numbers, letters, underscores, dashes and dots) and an optional oid.

The oid is required if you want to map a metric to *The embedded SNMP server*

9.6.2 Metric types

Before dealing with metrics you need to understand the various types represented by each metric:

COUNTER (type 0)

this is a generally-growing up number (like the number of requests)

GAUGE (type 1)

this is a number that can increase or decrease dinamically (like the memory used by a worker)

ABSOLUTE (type 2)

this is an absolute number, like the memory of the whole server, or the size of the hard disk.

ALIAS (type 3)

this is a virtual metric pointing to another one (you can use it to give different names to already existent metrics)

9.6.3 Metric collectors

Once you define a metric type, you need to tell uWSGI how to 'collect' the specific metric.

There are various ''collectors" available (and new can be added via plugins)

"ptr"

the value is collected from a memory pointer

"file"

the value is collected from a file

"sum"

the value is the sum of other metrics

"avg"

added in 1.9.20

compute the math average of the children

"accumulator"

always add the sum of children to the final value.

Ex:

round1: child1 = 22, child2 = 17 -> metric_value = 39

round2: child1 = 26, child2 = 30 -> metric_value += 56

'multiplier"

multiply the sum of children for the specified arg1n

child1 = 22, child2 = 17, arg1n = 3 -> metric_value = (22+17)*3

"func"

the value is computed calling a specific function every time

"manual" (the NULL collector)

the value must be updated manually from applications using the metrics api

9.6.4 Custom metrics

You can define additional metrics you can manage from your app.

The `--metric` option allows you to add more metrics.

It has a double syntax: simplified and keyval

```
uwsgi --http-socket :9090 --metric foobar
```

will create a metric 'foobar' with type 'counter', manual collector and no oid.

For creating advanced metrics you need the keyval way:

```
uwsgi --http-socket :9090 --metric name=foobar,type=gauge,oid=100.100.100
```

The following keys are available:

`name` set the metric name

`oid` set the metric oid

`type` set the metric type, can be `counter`, `gauge`, `absolute`, `alias`

`initial_value` set the metric to a specific value on startup

`freq` set the collection frequency in seconds (default to 1)

`children` maps children to the metric (see below)

`alias` the metric will be a simple alias for the specified one (–metric name=foobar,alias=worker.0.requests,type=alias)

`arg1` .. `arg3` string based arguments (see below)

`arg1n` .. `arg3n` number based arguments (see below)

`collector` set the collector, can be `ptr`, `file`, `sum`, `func` or anything exposed by plugins. Not specifying a collector means the metric is manual (your app needs to update it).

The ptr is currently unimplemented, while the other collector requires a bit of additional configuration:

`collector=file` requires `arg1` for the filename and an optional `arg1n` for the so-called split value.

```
uwsgi --metric name=loadavg,type=gauge,collector=file,arg1=/proc/loadavg,arg1n=1,freq=3
```

this will add a 'loadavg' metric, of type gauge, updated every 3 seconds with the content of /proc/loadavg. The content is splitted (using \n, \t, spaces, \r and zero as separator) and the item 1 (the returned array is zero-based) used as value.

the splitter is very powerful, so you could gather infos from more complex files, like /proc/meminfo

```
uwsgi --metric name=memory,type=gauge,collector=file,arg1=/proc/meminfo,arg1n=4,freq=3
```

once splitted, the /proc/meminfo has the MemFree value in the 4th slot

`collector=sum` requires the list of metrics that must be summed up. Each metric has the concept of 'children'. The sum collector will sum the values of all of its children:

```
uwsgi --metric name=reqs,collector=sum,children=worker.1.requests;worker.2.requests
```

this will sum the value of worker.1.requests and worker.2.requests every second

`collector=func` is a commodity collector avoiding you to write a whole plugin for adding a new collector.

Let's define a C function (call the file mycollector.c or whatever you want):

```
int64_t my_collector(void *metric) {
        return 173;
}
```

and build it as a shared library

```
gcc -shared -o mycollector.so mycollector.c
```

now run uWSGI

```
uwsgi --dlopen ./mycollector.so --metric name=mine,collector=func,arg1=my_collector,freq=10
```

this will call the C function my_collector every 10 seconds and will set the value of the metric 'mine' to its return value.

The function must returns an int64_t value. The argument it takes is a uwsgi_metric pointer. You generally do not need to parse it, so casting to void will avoid headaches.

9.6.5 The metrics directory

UNIX sysadmins love text files. They are generally the things they have to work on most of the time. If you want to make a UNIX sysadmin happy, just give him some text file to play with.

The metrics subsystem can expose all of its metrics in the form of text files in a directory:

```
uwsgi --metrics-dir mymetrics ...
```

(the mymetric dir must exists)

this will create a text file for each metric in the 'mymetrics' directory. The content of each file is the value of the metric (updated in realtime).

Each file is mapped in the process address space, so do not worry if your virtual memory increases.

9.6.6 Restoring metrics (persistent metrics)

When you restart a uWSGI instance, all of its metrics are reset.

This is generally the best thing to do, but if you want you can restore the previous situation, abusing the values stored in the metrics directory defined before.

Just add the `--metrics-dir-restore` option to force the metric subsystem to read-back the values from the metric directory before starting collecting values.

9.6.7 API

Your language plugins should expose at least the following api functions. Currently they are implemented in Perl, CPython, PyPy and Ruby

metric_get(name)

metric_set(name, value)

metric_inc(name[, delta])

metric_dec(name[, delta])

metric_mul(name[, delta])

metric_div(name[, delta])

metrics (tuple/array of metric keys, should be immutable and not-callable, currently unimplemented)

9.6.8 Stats pushers

Collected metrics can be sent to external systems for analysis or graphs generation.

Stats pushers are plugins aimed at sending metrics to those systems.

There are two kinds of stats pusher: json and raw.

json stats pusher send the whole json stats blob (the same you get from the stats server), while 'raw' ones send the metrics list.

Currently available stats pushers:

rrdtool

type: raw

plugin: rrdtool (builtin by default)

requires: librrd.so (dynamically detected on startup, not needed when building)

this will store an rrd file for each metric in the specified directory. Eacch rrd file has a single data source named 'metric'

Usage:

```
uwsgi --rrdtool my_rrds ...
```

or

```
uwsgi --stats-push rrdtool:my_rrds ...
```

by default the rrd files are updated every 300 seconds, you can tune this value with `--rrdtool-freq`

The librrd.so library is detected at runtime. If you need you can specify its absolute path with `--rrdtool-lib`

statsd

type: raw

plugin: stats_pusher_statsd

push metrics to a statsd server

syntax: –stats-push statsd:address[,prefix]

example:

```
uwsgi --stats-push statsd:127.0.0.1:8125,myinstance ...
```

carbon

type: raw

plugin: carbon (builtin by default)

see *Integration with Graphite/Carbon*

zabbix

type: raw

plugin: zabbix

push metrics to a zabbix server

syntax: –stats-push zabbix:address[,prefix]

example:

```
uwsgi --stats-push zabbix:127.0.0.1:10051,myinstance ...
```

The plugin exposes a `--zabbix-template` option that will generate a zabbix template (on stdout or in the specified file) containing all of the exposed metrics as trapper items.

Note: on some zabbox version you need to authorize the ip addresses allowed to push items

mongodb

type: json

plugin: stats_pusher_mongodb

required: libmongoclient.so

push statistics (as json) the the specified mongodb database

syntax (keyval): –stats-push mongodb:addr=<addr>,collection=<db>,freq=<freq>

file

type: json

plugin: stats_pusher_file

example plugin storing stats json in a file

socket

type: raw

plugin: stats_pusher_socket (builtin by default)

push metrics to a udp server with the following format:

<metric> <type> <value>

(<type> is in the numeric form previously reported)

syntax: –stats-push socket:address[,prefix]

Example:

```
uwsgi --stats-push socket:127.0.0.1:8125,myinstance ...
```

9.6.9 Alarms/Thresholds

You can configure one or more "thresholds" to each metric.

Once this limit is reached the specified alarm (see *The uWSGI alarm subsystem (from 1.3)*) is triggered.

Once the alarm is delivered you may choose to reset the counter to aspecfic value (generally 0), or continue triggering alarms with a specified rate.

```
[uwsgi]
...
metric-alarm = key=worker.0.avg_response_time,value=2000,alarm=overload,rate=30
metric-alarm = key=loadavg,value=3,alarm=overload,rate=120
metric-threshold = key=mycounter,value=1000,reset=0
...
```

Specifying an alarm is not required, using the threshold value to automatically reset a metric is perfectly valid

Note: –metric-threshold and –metric-alarm are the same option

9.6.10 SNMP integration

The *The embedded SNMP server* server exposes metrics starting from the 1.3.6.1.4.1.35156.17.3 OID.

For example to get the value of worker.0.requests:

```
snmpget -v2c -c <snmp_community> <snmp_addr>:<snmp_port> 1.3.6.1.4.1.35156.17.3.0.1
```

Remember: only metrics with an associated OID can be used via SNMP

9.6.11 Internal Routing integration

The ''router_metrics" plugin (builtin by default) adds a series of actions to the internal routing subsystem.

`metricinc:<metric>[,value]` increase the <metric>

`metricdec:<metric>[,value]` decrease the <metric>

`metricmul:<metric>[,value]` multiply the <metric>

`metricdiv:<metric>[,value]` divide the <metric>

`metricset:<metric>,<value>` set <metric> to <value>

in addition to action a route var named "metric" is added

Example:

```
[uwsgi]
metric = mymetric
route = ^/foo metricinc:mymetric
route-run = log:the value of the metric 'mymetric' is ${metric[mymetric]}
log-format = %(time) - %(metric.mymetric)
```

9.6.12 Request logging

You can access metrics values from your request logging format using the %(metric.xxx) placeholder:

```
[uwsgi]
log-format = [hello] %(time) %(metric.worker.0.requests)
```

9.6.13 Officially Registered Metrics

This is a work in progress, best way to know which default metrics are exposed is enabling the stats server and querying it (or adding the –metrics-dir option)

- worker/3 (exports information about workers, example worker.1.requests [or 3.1.1] reports the number of requests served by worker 1)
- plugin/4 (namespace for metrics automatically added by plugins, example plugins.foo.bar)
- core/5 (namespace for general instance informations)
- router/6 (namespace for corerouters, example router.http.active_sessions)
- socket/7 (namespace for sockets, example socket.0.listen_queue)
- mule/8 (namespace for mules, example mule.1.signals)
- spooler/9 (namespace for spoolers, example spooler.1.signals)
- system/10 (namespace for system metrics, like loadavg or free memory)

9.6.14 OID assigment for plugins

If you want to write plugin that will expose metrics, please first add OID namespace that you are going to use to the list below and make a pull request. This will ensure that all plugins are using unique OID namespaces. Prefix all plugin metric names with plugin name to ensure no conflicts if same keys are used in multiple plugins (example plugin.myplugin.foo.bar, worker.1.plugin.myplugin.foo.bar)

- (3|4).100.1 - cheaper_busyness

9.6.15 External tools

Check https://github.com/unbit/unbit-bars

CHAPTER 10

Async and loop engines

10.1 uWSGI asynchronous/non-blocking modes (updated to uWSGI 1.9)

Warning: Beware! Async modes will not speed up your app, they are aimed at improving concurrency. Do not expect that enabling some of the modes will work flawlessly, asynchronous/evented/non-blocking systems require app cooperation, so if your app is developed without taking specific async engine rules into consideration, you are doing it wrong. Do not trust people suggesting you to blindly use async/evented/non-blocking systems!

10.1.1 Glossary

uWSGI, following its modular approach, splits async engines into two families.

Suspend/Resume engines

They simply implement coroutine/green threads techniques. They have no event engine, so you have to use the one supplied by uWSGI. An Event engine is generally a library exporting primitives for platform-independent non-blocking I/O (libevent, libev, libuv, etc.). The uWSGI event engine is enabled using the `--async <n>` option.

Currently the uWSGI distribution includes the following suspend/resume engines:

- `uGreen` - Unbit's green thread implementation (based on `swapcontext()`)
- `Greenlet` - Python greenlet module
- `Stackless` - Stackless Python
- `Fiber` - Ruby 1.9 fibers

Running the uWSGI async mode without a proper suspend/resume engine will raise a warning, so for a minimal non-blocking app you will need something like that:

```
uwsgi --async 100 --ugreen --socket :3031
```

An important aspect of suspend/resume engines is that they can easily destroy your process if it is not aware of them. Some of the language plugins (most notably Python) have hooks to cooperate flawlessly with coroutines/green threads. Other languages may fail miserably. Always check the uWSGI mailing list or IRC channel for updated information.

Older uWSGI releases supported an additional system: callbacks. Callbacks is the approach used by popular systems like node.js. This approach requires **heavy** app cooperation, and for complex projects like uWSGI dealing with this is

extremely complex. For that reason, callback approach **is not supported** (even if technically possible) Software based on callbacks (like *The Tornado loop engine*) can be used to combine them with some form of suspend engine.

I/O engines (or event systems)

uWSGI includes an highly optimized evented technology, but can use alternative approaches too.

I/O engines always require some suspend/resume engine, otherwise ugly things happen (the whole uWSGI codebase is coroutine-friendly, so you can play with stacks pretty easily).

Currently supported I/O engines are:

- *The Tornado loop engine*
- `libuv` (work in progress)
- `libev` (work in progress)

Loop engines

Loop engines are packages/libraries exporting both suspend/resume techniques and an event system. When loaded, they override the way uWSGI manages connections and signal handlers (uWSGI signals, *not* POSIX signals).

Currently uWSGI supports the following loop engines:

- `Gevent` (Python, libev, greenlet)
- `Coro::AnyEvent` (Perl, coro, anyevent)

Although they are generally used by a specific language, pure-C uWSGI plugins (like the CGI one) can use them to increase concurrency without problems.

10.1.2 Async switches

To enable async mode, you use the `--async` option (or some shortcut for it, exported by loop engine plugins).

The argument of the `--async` option is the number of "cores" to initialize. Each core can manage a single request, so the more core you spawn, more requests you will be able to manage (*and more memory you will use*). The job of the suspend/resume engines is to stop the current request management, move to another core, and eventually come back to the old one (and so on).

Technically, cores are simple memory structures holding request's data, but to give the user the illusion of a multi-threaded system we use that term.

The switch between cores needs app cooperation. There are various ways to accomplish that, and generally, if you are using a loop engine, all is automagic (or requires very little effort).

> **Warning:** If you are in doubt, **do not use async mode**.

10.1.3 Running uWSGI in Async mode

To start uWSGI in async mode, pass the `--async` option with the number of "async cores" you want.

```
./uwsgi --socket :3031 -w tests.cpubound_async --async 10
```

This will start uWSGI with 10 async cores. Each async core can manage a request, so with this setup you can accept 10 concurrent requests with only one process. You can also start more processes (with the `--processes` option), each will have its own pool of async cores.

When using *harakiri* mode, every time an async core accepts a request, the harakiri timer is reset. So even if a request blocks the async system, harakiri will save you.

The `tests.cpubound_async` app is included in the source distribution. It's very simple:

```
def application(env, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    for i in range(1, 10000):
        yield "<h1>%s</h1>" % i
```

Every time the application does `yield` from the response function, the execution of the app is stopped, and a new request or a previously suspended request on another async core will take over. This means the number of async cores is the number of requests that can be queued.

If you run the `tests.cpubound_async` app on a non-async server, it will block all processing: will not accept other requests until the heavy cycle of 10000 `<h1>`s is done.

10.1.4 Waiting for I/O

If you are not under a loop engine, you can use the uWSGI API to wait for I/O events.

Currently only 2 functions are exported:

- `uwsgi.wait_fd_read()`
- `uwsgi.wait_fd_write()`

These functions may be called in succession to wait for multiple file descriptors:

```
uwsgi.wait_fd_read(fd0)
uwsgi.wait_fd_read(fd1)
uwsgi.wait_fd_read(fd2)
yield ""  # yield the app, let uWSGI do its magic
```

10.1.5 Sleeping

On occasion you might want to sleep in your app, for example to throttle bandwidth.

Instead of using the blocking `time.sleep(N)` function, use `uwsgi.async_sleep(N)` to yield control for N seconds.

See also:

See `tests/sleeping_async.py` for an example.

10.1.6 Suspend/Resume

Yielding from the main application routine is not very practical, as most of the time your app is more advanced than a simple callable and is formed of tons of functions and various levels of call depth.

Worry not! You can force a suspend (using coroutine/green thread) by simply calling `uwsgi.suspend()`:

```
uwsgi.wait_fd_read(fd0)
uwsgi.suspend()
```

`uwsgi.suspend()` will automatically call the chosen suspend engine (uGreen, greenlet, etc.).

10.1.7 Static files

Static file server will automatically use the loaded async engine.

10.2 The Gevent loop engine

Gevent is an amazing non-blocking Python network library built on top of `libev` and `greenlet`. Even though uWSGI supports Greenlet as suspend-resume/greenthread/coroutine library, it requires a lot of effort and code modifications to work with gevent. The gevent plugin requires gevent 1.0.0 and *uWSGI asynchronous/non-blocking modes (updated to uWSGI 1.9)* mode.

10.2.1 Notes

- The `SignalFramework` is fully working with Gevent mode. Each handler will be executed in a dedicated greenlet. Look at `tests/ugevent.py` for an example.
- uWSGI multithread mode (`threads` option) will not work with Gevent. Running Python threads in your apps is supported.
- Mixing uWSGI's Async API with gevent's is **EXPLICITLY FORBIDDEN**.

10.2.2 Building the plugin (uWSGI >= 1.4)

The gevent plugin is compiled in by default when the default profile is used. Doing the following will install the python plugin as well as the gevent one:

```
pip install uwsgi
```

10.2.3 Building the plugin (uWSGI < 1.4)

A 'gevent' build profile can be found in the `buildconf` directory.

```
python uwsgiconfig --build gevent
# or...
UWSGI_PROFILE=gevent make
# or...
UWSGI_PROFILE=gevent pip install git+git://github.com/unbit/uwsgi.git
# or...
python uwsgiconfig --plugin plugins/gevent # external plugin
```

10.2.4 Running uWSGI in gevent mode

```
uwsgi --gevent 100 --socket :3031 --module myapp
```

or for a modular build:

```
uwsgi --plugins python,gevent --gevent 100 --socket :3031 --module myapp
```

the argument of –gevent is the number of async cores to spawn

10.2.5 A crazy example

The following example shows how to sleep in a request, how to make asynchronous network requests and how to continue doing logic after a request has been closed.

```
import gevent
import gevent.socket

def bg_task():
    for i in range(1,10):
        print "background task", i
        gevent.sleep(2)

def long_task():
    for i in range(1,10):
        print i
        gevent.sleep()

def application(e, sr):
    sr('200 OK', [('Content-Type','text/html')])
    t = gevent.spawn(long_task)
    t.join()
    yield "sleeping for 3 seconds...<br/>"
    gevent.sleep(3)
    yield "done<br>"
    yield "getting some ips...<br/>"
    urls = ['www.google.com', 'www.example.com', 'www.python.org', 'projects.unbit.it']
    jobs = [gevent.spawn(gevent.socket.gethostbyname, url) for url in urls]
    gevent.joinall(jobs, timeout=2)

    for j in jobs:
        yield "ip = %s<br/>" % j.value

    gevent.spawn(bg_task) # this task will go on after request end
```

10.2.6 Monkey patching

uWSGI uses native gevent api, so it does not need monkey patching. That said, your code may need it, so remember to call `gevent.monkey.patch_all()` at the start of your app. As of uWSGI 1.9, the convenience option `--gevent-monkey-patch` will do that for you.

A common example is using `psycopg2_gevent` with django. Django will make a connection to postgres for each thread (storing it in thread locals).

As the uWSGI gevent plugin runs on a single thread this approach will lead to a deadlock in psycopg. Enabling monkey patch will allow you to map thread locals to greenlets (though you could avoid full monkey patching and only call `gevent.monkey.patch_thread()`) and solves the issue:

```
import gevent.monkey
gevent.monkey.patch_thread()
import gevent_psycopg2
gevent_psycopg2.monkey_patch()
```

or (to monkey patch everything)

```
import gevent.monkey
gevent.monkey.patch_all()
```

```
import gevent_psycopg2
gevent_psycopg2.monkey_patch()
```

10.2.7 Notes on clients and frontends

- If you're testing a WSGI application that generates a stream of data, you should know that `curl` by default buffers data until a newline. So make sure you either disable curl's buffering with the `-N` flag or have regular newlines in your output.
- If you are using Nginx in front of uWSGI and wish to stream data from your app, you'll probably want to disable Nginx's buffering.

```
uwsgi_buffering off;
```

10.3 The Tornado loop engine

Available from: `'uWSGI 1.9.19-dev'`

Supported suspend engines: `'greenlet'`

Supported CPython versions: `'all of tornado supported versions'`

The tornado loop engine allows you to integrate your uWSGI stack with the Tornado IOLoop class.

Basically every I/O operation of the server is mapped to a tornado IOLoop callback. Making RPC, remote caching, or simply writing responses is managed by the Tornado engine.

As uWSGI is not written with a callback-based programming approach, integrating with those kind of libraries requires some form of "suspend" engine (green threads/coroutines)

Currently the only supported suspend engine is the "greenlet" one. Stackless python could work too (needs testing).

PyPy is currently not supported (albeit technically possibile thanks to continulets). Drop a mail to Unbit staff if you are interested.

10.3.1 Why ?

The Tornado project includes a simple WSGI server by itself. In the same spirit of the Gevent plugin, the purpose of Loop engines is allowing external prejects to use (and abuse) the uWSGI api, for better performance, versatility and (maybe the most important thing) resource usage.

All of the uWSGI subsystems are available (from caching, to websockets, to metrics) in your tornado apps, and the WSGI engine is the battle-tested uWSGI one.

10.3.2 Installation

The tornado plugin is currently not built-in by default. To have both tornado and greenlet in a single binary you can do

```
UWSGI_EMBED_PLUGINS=tornado,greenlet pip install tornado greenlet uwsgi
```

or (from uWSGI sources, if you already have tornado and greenlet installed)

```
UWSGI_EMBED_PLUGINS=tornado,greenlet make
```

10.3.3 Running it

The `--tornado` option is exposed by the tornado plugin, allowing you to set optimal parameters:

```
uwsgi --http-socket :9090 --wsgi-file myapp.py --tornado 100 --greenlet
```

this will run a uWSGI instance on http port 9090 using tornado as I/O (and time) management and greenlet as suspend engine

100 async cores are allocated, allowing you to manage up to 100 concurrent requests

10.3.4 Integrating WSGI with the tornado api

For the way WSGI works, dealing with callback based programming is pretty hard (if not impossible).

Thanks to greenlet we can suspend the execution of our WSGI callable until a tornado IOLoop event is available:

```
from tornado.httpclient import AsyncHTTPClient
import greenlet
import functools

# this gives us access to the main IOLoop (the same used by uWSGI)
from tornado.ioloop import IOLoop
io_loop = IOLoop.instance()

# this is called at the end of the external HTTP request
def handle_request(me, response):
    if response.error:
        print("Error:", response.error)
    else:
        me.result = response.body
    # back to the WSGI callable
    me.switch()

 def application(e, sr):
     me = greenlet.getcurrent()
     http_client = AsyncHTTPClient()
     http_client.fetch("http://localhost:9191/services", functools.partial(handle_request, me))
     # suspend the execution until an IOLoop event is available
     me.parent.switch()
     sr('200 OK', [('Content-Type','text/plain')])
     return me.result
```

10.3.5 Welcome to Callback-Hell

As always, it is not the job of uWSGI to judge programming approaches. It is a tool for sysadmins, and sysadmins should be tolerant with developers choices.

One of the things you will pretty soon experiment with this approach to programming is the callback-hell.

Let's extend the previous example to wait 10 seconds before sending back the response to the client

```
from tornado.httpclient import AsyncHTTPClient
import greenlet
import functools

# this gives us access to the main IOLoop (the same used by uWSGI)
from tornado.ioloop import IOLoop
```

```
io_loop = IOLoop.instance()

def sleeper(me):
    #TIMED OUT
    # finally come back to WSGI callable
    me.switch()

# this is called at the end of the external HTTP request
def handle_request(me, response):
    if response.error:
        print("Error:", response.error)
    else:
        me.result = response.body
    # add another callback in the chain
    me.timeout = io_loop.add_timeout(time.time() + 10, functools.partial(sleeper, me))

 def application(e, sr):
     me = greenlet.getcurrent()
     http_client = AsyncHTTPClient()
     http_client.fetch("http://localhost:9191/services", functools.partial(handle_request, me))
     # suspend the execution until an IOLoop event is available
     me.parent.switch()
     # unregister the timer
     io_loop.remove_timeout(me.timeout)
     sr('200 OK', [('Content-Type','text/plain')])
     return me.result
```

here we have chained two callbacks, with the last one being responsable for giving back control to the WSGI callable

The code could looks ugly or overcomplex (compared to other approaches like gevent) but this is basically the most efficient way to increase concurrency (both in terms of memory usage and performance). Technologies like node.js are becoming popular thanks to the results they allow to accomplish.

10.3.6 WSGI generators (aka yield all over the place)

Take the following WSGI app:

```
def application(e, sr):
    sr('200 OK', [('Content-Type','text/html')])
    yield "one"
    yield "two"
    yield "three"
```

if you have already played with uWSGI async mode, you knows that every yield internally calls the used suspend engine (greenlet.switch() in our case).

That means we will enter the tornado IOLoop engine soon after having called "application()". How we can give the control back to our callable if we are not waiting for events ?

The uWSGI async api has been extended to support the "schedule_fix" hook. It allows you to call a hook soon after the suspend engine has been called.

In the tornado's case this hook is mapped to something like:

```
io_loop.add_callback(me.switch)
```

in this way after every yield a me.switch() function is called allowing the resume of the callable.

Thanks to this hook you can transparently host standard WSGI applications without changing them.

10.3.7 Binding and listening with Tornado

The Tornado IOLoop is executed after fork() in every worker. If you want to bind to network addresses with Tornado, remember to use different ports for each workers:

```
from uwsgidecorators import *
import tornado.web

# this is our Tornado-managed app
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

t_application = tornado.web.Application([
    (r"/", MainHandler),
])

# here happens the magic, we bind after every fork()
@postfork
def start_the_tornado_servers():
    application.listen(8000 + uwsgi.worker_id())

# this is our WSGI callable managed by uWSGI
def application(e, sr):
    ...
```

Remember: do no start the IOLoop class. uWSGI will do it by itself as soon as the setup is complete

10.4 uGreen – uWSGI Green Threads

uGreen is an implementation of green threads on top of the *uWSGI async platform*.

It is very similar to Python's greenlet but built on top of the POSIX `swapcontext()` function. To take advantage of uGreen you have to set the number of async cores that will be mapped to green threads.

For example if you want to spawn 30 green threads:

```
./uwsgi -w tests.cpubound_green -s :3031 --async 30 --ugreen
```

The `ugreen` option will enable uGreen on top of async mode.

Now when you call `uwsgi.suspend()` in your app, you'll be switched off to another green thread.

10.4.1 Security and performance

To ensure (relative) isolation of green threads, every stack area is protected by so called "guard pages".

An attempt to write out of the stack area of a green thread will result in a segmentation fault/bus error (and the process manager, if enabled, will respawn the worker without too much damage).

The context switch is very fast, we can see it as:

- On switch
  1. Save the Python Frame pointer
  2. Save the recursion depth of the Python environment (it is simply an int)
  3. Switch to the main stack

- On return
  1. Re-set the uGreen stack
  2. Re-set the recursion depth
  3. Re-set the frame pointer

The stack/registers switch is done by the POSIX `swapcontext()` call and we don't have to worry about it.

10.4.2 Async I/O

For managing async I/O you can use the Async mode FD wait functions `uwsgi.wait_fd_read()` and `uwsgi.wait_fd_write()`.

10.4.3 Stack size

You can choose the uGreen stack size using the `ugreen-stacksize <pages>` option. The argument is in pages, not bytes.

10.4.4 Is this better than Greenlet or Stackless Python?

Weeeeelll... it depends. uGreen is faster (the stack is preallocated) but requires more memory (to allocate a stack area for every core). Stackless and Greenlet probably require less memory... but Stackless requires a heavily patched version of Python.

If you're heavily invested in making your app as async-snappy as possible, it's always best to do some tests to choose the best one for you. As far as uWSGI is concerned, you can move from async engine to another without changing your code.

10.4.5 What about `python-coev`?

Lots of uGreen has been inspired by it. The author's way to map Python threads to their implementation allows `python-coev` to be a little more "trustworthy" than Stackless Python. However, like Stackless, it requires a patched version of Python... :(

10.4.6 Can I use uGreen to write Comet apps?

Yeah! Sure! Go ahead. In the distribution you will find the `ugreenchat.py` script. It is a simple/dumb multiuser Comet chat. If you want to test it (for example 30 users) run it with

```
./uwsgi -s :3031 -w ugreenchat --async 30 --ugreen
```

The code has comments for every ugreen-related line. You'll need Bottle, an amazing Python web micro framework to use it.

10.4.7 Psycopg2 improvements

uGreen can benefit from the new psycopg2 async extensions and the psycogreen project. See the `tests/psycopg2_green.py` and `tests/psycogreen_green.py` files for examples.

CHAPTER 11

Web Server support

11.1 Apache support

Currently there are three uwsgi-protocol related apache2 modules available.

11.1.1 mod_uwsgi

This is the original module. It is solid, but incredibly ugly and does not follow a lot of apache coding convention style.

`mod_uwsgi` can be used in two ways:

- The "assbackwards" way (the default one). It is the fastest but somewhat far from the Apache2 API. If you do not use Apache2 filters (including gzip) for content generated by uWSGI, use this mode.
- The "cgi" mode. This one is somewhat slower but better integrated with Apache. To use the CGI mode, pass `-C` to the uWSGI server.

Options

Note: All of the options can be set per-host or per-location.

uWSGISocket <path> [timeout] Absolute path and optional timeout in seconds of uwsgi server socket.

uWSGISocket2 <path> Absolute path of failover uwsgi server socket

uWSGIServer <host:port> Address and port of an UWSGI server (e.g. localhost:4000)

uWSGIModifier1 <int> Set uWSGI modifier1

uWSGIModifier2 <int> Set uWSGI modifier2

uWSGIForceScriptName <value> Force `SCRIPT_NAME` (app name)

uWSGIForceCGIMode <on/off> Force uWSGI CGI mode for perfect integration with apache filters

uWSGIForceWSGIScheme <value> Force the WSGI scheme var (set by default to "http")

uWSGIMaxVars <int> Set the maximum allowed number of uwsgi protocol variables (default 128)

To pass custom variables use the `SetEnv` directive:

```
SetEnv UWSGI_SCRIPT yourapp
```

11.1.2 mod_proxy_uwsgi

This is the latest module and probably the best bet for the future. It is a "proxy" module, so you will get all of the features exported by mod_proxy. It is fully "apache api compliant" so it should be easy to integrate with the available modules. Using it is easy; just remember to load mod_proxy and mod_proxy_uwsgi modules in your apache config.

```
ProxyPass /foo uwsgi://127.0.0.1:3032/
ProxyPass /bar uwsgi://127.0.0.1:3033/
ProxyPass / uwsgi://127.0.0.1:3031/
```

The first two forms set SCRIPT_NAME respectively to /foo and /bar while the last one use an empty SCRIPT_NAME. You can set additional uwsgi vars using the SetEnv directive and load balance requests using mod_proxy_balancer.

```
<Proxy balancer://mycluster>
   BalancerMember uwsgi://192.168.1.50:3031/
   BalancerMember uwsgi://192.168.1.51:3031/
</Proxy>
ProxyPass / balancer://mycluster
```

Pay attention to the last slash in the member/node definition. It is optional for non-empty SCRIPT_NAME/mountpoints but required for apps mounted in the root of the domain. Currently the module lacks the ability to set modifiers, though this will be fixed soon.

Note: If you want to use this module (and help the uWSGI project), report any bugs you find, rather than falling back to the ancient (and ugly) mod_uwsgi

---

11.1.3 mod_Ruwsgi

This module is based on the SCGI module written by Roger Florkowski.

Note: This module is currently undocumented.

11.2 Cherokee support

Note: Recent official versions of Cherokee have an uWSGI configuration wizard. If you want to use it you have to install uWSGI in a directory included in your system `PATH`.

- Set the UWSGI handler for your target.
- If you are using the default target (`/`) remember to uncheck the `check_file` property.
- Configure an "information source" of type "Remote", specifying the socket name of uWSGI. If your uWSGI has TCP support, you can build a cluster by spawning the uWSGI server on a different machine.

Note: Remember to add a target for all of your URI containing static files (ex. /media /images ...) using an appropriate handler

---

11.2.1 Dynamic apps

If you want to hot-add apps specify the `UWSGI_SCRIPT` var in the uWSGI handler options:

- In the section: *Add new custom environment variable* specify `UWSGI_SCRIPT` as name and the name of your WSGI script (without the .py extension) as the value.

Your app will be loaded automatically at the first request.

11.3 Native HTTP support

11.3.1 HTTPS support (from 1.3)

Use the `https <socket>,<certificate>,<key>` option. This option may be specified multiple times. First generate your server key, certificate signing request, and self-sign the certificate using the OpenSSL toolset:

Note: You'll want a real SSL certificate for production use.

```
openssl genrsa -out foobar.key 2048
openssl req -new -key foobar.key -out foobar.csr
openssl x509 -req -days 365 -in foobar.csr -signkey foobar.key -out foobar.crt
```

Then start the server using the SSL certificate and key just generated:

```
uwsgi --master --https 0.0.0.0:8443,foobar.crt,foobar.key
```

As port 443, the port normally used by HTTPS, is privileged (ie. non-root processes may not bind to it), you can use the shared socket mechanism and drop privileges after binding like thus:

```
uwsgi --shared-socket 0.0.0.0:443 --uid roberto --gid roberto --https =0,foobar.crt,foobar.key
```

uWSGI will bind to 443 on any IP, then drop privileges to those of `roberto`, and use the shared socket 0 (`=0`) for HTTPS.

Note: The =0 syntax is currently undocumented.

Setting SSL/TLS ciphers

The `https` option takes an optional fourth argument you can use to specify the OpenSSL cipher suite.

```
[uwsgi]
master = true
shared-socket = 0.0.0.0:443
uid = www-data
gid = www-data

https = =0,foobar.crt,foobar.key,HIGH
http-to = /tmp/uwsgi.sock
```

This will set all of the **HIGHest** ciphers (whenever possible) for your SSL/TLS transactions.

Client certificate authentication

The `https` option can also take an optional 5th argument. You can use it to specify a CA certificate to authenticate your clients with. Generate your CA key and certificate (this time the key will be 4096 bits and password-protected):

```
openssl genrsa -des3 -out ca.key 4096
openssl req -new -x509 -days 365 -key ca.key -out ca.crt
```

Generate the server key and CSR (as before):

```
openssl genrsa -out foobar.key 2048
openssl req -new -key foobar.key -out foobar.csr
```

Sign the server certificate with your new CA:

```
openssl x509 -req -days 365 -in foobar.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out foobar.crt
```

Create a key and a CSR for your client, sign it with your CA and package it as PKCS#12. Repeat these steps for each client.

```
openssl genrsa -des3 -out client.key 1024
openssl req -new -key client.key -out client.csr
openssl x509 -req -days 365 -in client.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out client.crt
openssl pkcs12 -export -in client.crt -inkey client.key -name "Client 01" -out client.p12
```

Then configure uWSGI for certificate client authentication

```
[uwsgi]
master = true
shared-socket = 0.0.0.0:443
uid = www-data
gid = www-data
https = =0,foobar.crt,foobar.key,HIGH,!ca.crt
http-to = /tmp/uwsgi.sock
```

Note: If you don't want the client certificate authentication to be mandatory, remove the '!' before ca.crt in the https options.

11.3.2 HTTP sockets

The `http-socket <bind>` option will make uWSGI natively speak HTTP. If your web server does not support the *uwsgi protocol* but is able to speak to upstream HTTP proxies, or if you are using a service like Webfaction or Heroku to host your application, you can use `http-socket`. If you plan to expose your app to the world with uWSGI only, use the `http` option instead, as the router/proxy/load-balancer will then be your shield.

11.3.3 The uWSGI HTTP/HTTPS router

uWSGI includes an HTTP/HTTPS router/proxy/load-balancer that can forward requests to uWSGI workers. The server can be used in two ways: embedded and standalone. In embedded mode, it will automatically spawn workers and setup the communication socket. In standalone mode you have to specify the address of a uwsgi socket to connect to.

Embedded mode:

```
./uwsgi --http 127.0.0.1:8080 --master --module mywsgiapp --processes 4
```

This will spawn a HTTP server on port 8080 that forwards requests to a pool of 4 uWSGI workers managed by the master process.

Standalone mode:

```
./uwsgi --master --http 127.0.0.1:8080 --http-to /tmp/uwsgi.sock
```

This will spawn a HTTP router (governed by a master for your safety) that will forward requests to the uwsgi socket `/tmp/uwsgi.sock`. You can bind to multiple addresses/ports.

```
[uwsgi]

http = 0.0.0.0:8080
http = 192.168.173.17:8181
http = 127.0.0.1:9090

master = true

http-to = /tmp/uwsgi.sock
```

And load-balance to multiple nodes:

```
[uwsgi]

http = 0.0.0.0:8080
http = 192.168.173.17:8181
http = 127.0.0.1:9090

master = true

http-to = /tmp/uwsgi.sock
http-to = 192.168.173.1:3031
http-to = 192.168.173.2:3031
http-to = 192.168.173.3:3031
```

- If you want to go massive (virtualhosting and zero-conf scaling) combine the HTTP router with the *uWSGI Subscription Server*.
- You can make the HTTP server pass custom uwsgi variables to workers with the `http-var KEY=VALUE` option.
- You can use the `http-modifier1` option to pass a custom *modifier1* value to workers.

11.3.4 HTTPS support

see *HTTPS support (from 1.3)*

11.3.5 HTTP Keep-Alive

If your backends set the correct HTTP headers, you can use the `http-keepalive` option. Your backends will need to set a valid `Content-Length` in each response or use chunked encoding. Simply setting "Connection: close" is *not enough*. Also remember to set "Connection: Keep-Alive" in your response. You can automate that using the `add-header "Connection: Keep-Alive"` option.

11.3.6 Can I use uWSGI's HTTP capabilities in production?

If you need a load balancer/proxy it can be a very good idea. It will automatically find new uWSGI instances and can load balance in various ways. If you want to use it as a real webserver you should take into account that serving static files in uWSGI instances is possible, but not as good as using a dedicated full-featured web server. If you host

static assets in the cloud or on a CDN, using uWSGI's HTTP capabilities you can definitely avoid configuring a full webserver.

Note: If you use Amazon's ELB (Elastic Load Balancer) in HTTP mode in front of uWSGI in HTTP mode, a valid `Content-Length` *must be set* by the backend.

11.4 HTTPS support (from 1.3)

Use the `https <socket>,<certificate>,<key>` option. This option may be specified multiple times. First generate your server key, certificate signing request, and self-sign the certificate using the OpenSSL toolset:

Note: You'll want a real SSL certificate for production use.

```
openssl genrsa -out foobar.key 2048
openssl req -new -key foobar.key -out foobar.csr
openssl x509 -req -days 365 -in foobar.csr -signkey foobar.key -out foobar.crt
```

Then start the server using the SSL certificate and key just generated:

```
uwsgi --master --https 0.0.0.0:8443,foobar.crt,foobar.key
```

As port 443, the port normally used by HTTPS, is privileged (ie. non-root processes may not bind to it), you can use the shared socket mechanism and drop privileges after binding like thus:

```
uwsgi --shared-socket 0.0.0.0:443 --uid roberto --gid roberto --https =0,foobar.crt,foobar.key
```

uWSGI will bind to 443 on any IP, then drop privileges to those of `roberto`, and use the shared socket 0 (`=0`) for HTTPS.

Note: The =0 syntax is currently undocumented.

11.4.1 Setting SSL/TLS ciphers

The `https` option takes an optional fourth argument you can use to specify the OpenSSL cipher suite.

```
[uwsgi]
master = true
shared-socket = 0.0.0.0:443
uid = www-data
gid = www-data

https = =0,foobar.crt,foobar.key,HIGH
http-to = /tmp/uwsgi.sock
```

This will set all of the **HIGHest** ciphers (whenever possible) for your SSL/TLS transactions.

11.4.2 Client certificate authentication

The `https` option can also take an optional 5th argument. You can use it to specify a CA certificate to authenticate your clients with. Generate your CA key and certificate (this time the key will be 4096 bits and password-protected):

```
openssl genrsa -des3 -out ca.key 4096
openssl req -new -x509 -days 365 -key ca.key -out ca.crt
```

Generate the server key and CSR (as before):

```
openssl genrsa -out foobar.key 2048
openssl req -new -key foobar.key -out foobar.csr
```

Sign the server certificate with your new CA:

```
openssl x509 -req -days 365 -in foobar.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out foobar.crt
```

Create a key and a CSR for your client, sign it with your CA and package it as PKCS#12. Repeat these steps for each client.

```
openssl genrsa -des3 -out client.key 1024
openssl req -new -key client.key -out client.csr
openssl x509 -req -days 365 -in client.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out client.crt
openssl pkcs12 -export -in client.crt -inkey client.key -name "Client 01" -out client.p12
```

Then configure uWSGI for certificate client authentication

```
[uwsgi]
master = true
shared-socket = 0.0.0.0:443
uid = www-data
gid = www-data
https = =0,foobar.crt,foobar.key,HIGH,!ca.crt
http-to = /tmp/uwsgi.sock
```

Note: If you don't want the client certificate authentication to be mandatory, remove the '!' before ca.crt in the https options.

11.5 The SPDY router (uWSGI 1.9)

Starting from uWSGI 1.9 the HTTPS router has been extended to support version 3 of the SPDY protocol.

To run the HTTPS router with SPDY support, use the `--https2` option:

```
uwsgi --https2 addr=0.0.0.0:8443,cert=foobart.crt,key=foobar.key,spdy=1 --module werkzeug.testapp:tes
```

This will start an HTTPS router on port 8443 with SPDY support, forwarding requests to the Werkzeug's test app the instance is running. If you'll go to https://address:8443/ with a SPDY-enabled browser, you will see additional WSGI variables reported by Werkzeug:

- `SPDY` – `on`
- `SPDY.version` – protocol version (generally `3`)
- `SPDY.stream` – stream identifier (an odd number).

11.5.1 Notes

- You need at least OpenSSL 1.x to use SPDY (all of the modern Linux distros should have it).
- During uploads, the window size is constantly updated.

- The `--http-timeout` directive is used to set the SPDY timeout. This is the maximum amount of inactivity after the SPDY connection is closed.
- `PING` requests from the browsers are **all** acknowledged.
- On connect, the SPDY router sends a settings packet to the client with optimal values.
- If a stream fails in some catastrophic way, the whole connection is closed hard.
- `RST` messages are always honoured.

11.5.2 TODO

- Add old SPDY v2 support (is it worth it?)
- Allow PUSHing of resources from the uWSGI cache
- Allow tuning internal buffers

11.6 Lighttpd support

Note: Lighttpd support is experimental.

The uwsgi handler for Lighttpd lives in the `/lighttpd` directory of the uWSGI distribution.

11.6.1 Building the module

First download the source of lighttpd and uncompress it. Copy the `lighttpd/mod_uwsgi.c` file from the uWSGI distribution into Lighttpd's `/src` directory. Add the following to the lighttpd `src/Makefile.am` file, after the accesslog block:

```
lib_LTLIBRARIES += mod_uwsgi.la
mod_uwsgi_la_SOURCES = mod_uwsgi.c
mod_uwsgi_la_LDFLAGS = -module -export-dynamic -avoid-version -no-undefined
mod_uwsgi_la_LIBADD = $(common_libadd)
```

Then launch

```
autoreconf -fi
```

and as usual,

```
./configure && make && make install
```

11.6.2 Configuring Lighttpd

Modify your configuration file:

```
server.modules = (
  ...
  "mod_uwsgi",
  ...
)
```

```
# ...

uwsgi.server = (
  "/pippo" => (( "host" => "192.168.173.15", "port" => 3033 )),
  "/" => (( "host" => "127.0.0.1", "port" => 3031 )),
)
```

If you specify multiple hosts under the same virtual path/URI, load balancing will be activated with the "Fair" algorithm.

11.7 Attaching uWSGI to Mongrel2

Mongrel2 is a next-next-generation webserver that focuses on modern webapps.

Just like uWSGI, it is fully language agnostic, cluster-friendly and delightfully controversial :)

It uses the amazing ZeroMQ library for communication, allowing reliable, easy message queueing and configuration-free scalability.

Starting from version 0.9.8-dev, uWSGI can be used as a Mongrel2 handler.

11.7.1 Requirements

To enable ZeroMQ/Mongrel2 support in uWSGI you need the zeromq library (2.1+) and the uuid library.

Mongrel2 can use JSON or tnetstring to pass data (such as headers and various other information) to handlers. uWSGI supports tnetstring out of the box but requires the Jansson library to parse JSON data. If you don't install jansson or do not want to use JSON, make sure you specify `protocol='tnetstring'` in the Handler in the Mongrel2 configuration, as the default is to use JSON. This would result in a rather obscure "JSON support not enabled. Skip request" message in the uWSGI log.

11.7.2 Configuring Mongrel2

You can find `mongrel2-uwsgi.conf` shipped with the uWSGI source. You can use this file as a base to configure Mongrel2.

```
main = Server(
    uuid="f400bf85-4538-4f7a-8908-67e313d515c2",
    access_log="/logs/access.log",
    error_log="/logs/error.log",
    chroot="./",
    default_host="192.168.173.11",
    name="test",
    pid_file="/run/mongrel2.pid",
    port=6767,
    hosts = [
        Host(name="192.168.173.11", routes={
            '/': Handler(send_spec='tcp://192.168.173.11:9999',
                    send_ident='54c6755b-9628-40a4-9a2d-cc82a816345e',
                    recv_spec='tcp://192.168.173.11:9998', recv_ident='',
                    protocol='tnetstring')
        })
    ]
)
```

```
settings = {'upload.temp_store':'tmp/mongrel2.upload.XXXXXX'}
servers = [main]
```

It is a pretty standard Mongrel2 configuration with upload streaming enabled.

11.7.3 Configuring uWSGI for Mongrel2

To attach uWSGI to Mongrel2, simply use the *zeromq, zmq, zeromq-socket, zmq-socket* option:

```
uwsgi --zeromq tcp://192.168.173.11:9999,tcp://192.168.173.11:9998
```

You can spawn multiple processes (each one will subscribe to Mongrel2 with a different uuid)

```
uwsgi --zeromq tcp://192.168.173.11:9999,tcp://192.168.173.11:9998 -p 4
```

You can use threads too. Each thread will subscribe to the Mongrel2 queue but the responder socket will be shared by all the threads and protected by a mutex.

```
uwsgi --zeromq tcp://192.168.173.11:9999,tcp://192.168.173.11:9998 -p 4 --threads 8
# This will spawn 4 processes with 8 threads each, totaling 32 threads.
```

11.7.4 Test them all

Add an application to uWSGI (we will use the werkzeug.testapp as always)

```
uwsgi --zeromq tcp://192.168.173.11:9999,tcp://192.168.173.11:9998 -p 4 --threads 8 --module werkzeug
```

Now launch the command on all the servers you want, Mongrel2 will distribute requests to them automagically.

11.7.5 Async mode

> **Warning:** Async support for ZeroMQ is still under development, as ZeroMQ uses edge triggered events that complicate things in the uWSGI async architecture.

11.7.6 Chroot

By default Mongrel2 will `chroot()`. This is a good thing for security, but can cause headaches regarding file upload streaming. Remember that Mongrel2 will save the uploaded file in its own chroot jail, so if your uWSGI instance does not live in the same chroot jail, you'll have to choose the paths carefully. In the example Mongrel2 configuration file we have used a relative path to easily allow uWSGI to reach the file.

11.7.7 Performance

Mongrel2 is extremely fast and reliable even under huge loads. tnetstring and JSON are text-based (so they are a little less effective than the binary *uwsgi protocol*. However, as Mongrel2 does not require the expensive one-connection-for-request method, you should get pretty much the same (if not higher) results compared to a (for example) *Nginx* + uWSGI approach.

11.7.8 uWSGI clustering + ZeroMQ

You can easily mix uWSGI `clustering` with ZeroMQ.

Choose the main node and run

```
uwsgi --zeromq tcp://192.168.173.11:9999,tcp://192.168.173.11:9998 -p 4 --threads 8 --module werkzeug
```

And on all the other nodes simply run

```
uwsgi --cluster 225.1.1.1:1717
```

11.7.9 Mixing standard sockets with ZeroMQ

You can add uwsgi/*HTTP*/FastCGI/... sockets to your uWSGI server in addition to ZeroMQ, but if you do, remember to disable threads! This limitation will probably be fixed in the future.

11.7.10 Logging via ZeroMQ

See also:

`ZeroMQLogging`

11.8 Nginx support

Nginx natively includes support for upstream servers speaking the *uwsgi protocol* since version 0.8.40.

If you are unfortunate enough to use an older version (that nevertheless is 0.7.63 or newer), you can find a module in the *nginx* directory of the uWSGI distribution.

11.8.1 Building the module (Nginx 0.8.39 and older)

Download a >=0.7.63 release of nginx and untar it at the same level of your uWSGI distribution directory. Move yourself into the nginx-0.7.x directory and `./configure` nginx to add the uwsgi handler to its module list:

```
./configure --add-module=../uwsgi/nginx/
```

then `make` and `make install` it.

If all goes well you can now configure Nginx to pass requests to the uWSGI server.

11.8.2 Configuring Nginx

First of all copy the `uwsgi_params` file (available in the *nginx* directory of the uWSGI distribution) into your Nginx configuration directory, then in a *location* directive in your Nginx configuration add:

```
uwsgi_pass unix:///tmp/uwsgi.sock;
include uwsgi_params;
```

– or if you are using TCP sockets,

```
uwsgi_pass 127.0.0.1:3031;
include uwsgi_params;
```

Then simply reload Nginx and you are ready to rock your uWSGI powered applications through Nginx.

What is the `uwsgi_params` file?

It's convenience, nothing more! For your reading pleasure, the contents of the file as of uWSGI 1.3:

```
uwsgi_param QUERY_STRING $query_string;
uwsgi_param REQUEST_METHOD $request_method;
uwsgi_param CONTENT_TYPE $content_type;
uwsgi_param CONTENT_LENGTH $content_length;
uwsgi_param REQUEST_URI $request_uri;
uwsgi_param PATH_INFO $document_uri;
uwsgi_param DOCUMENT_ROOT $document_root;
uwsgi_param SERVER_PROTOCOL $server_protocol;
uwsgi_param REMOTE_ADDR $remote_addr;
uwsgi_param REMOTE_PORT $remote_port;
uwsgi_param SERVER_ADDR $server_addr;
uwsgi_param SERVER_PORT $server_port;
uwsgi_param SERVER_NAME $server_name;
```

See also:

uwsgi protocol magic variables

11.8.3 Clustering

Nginx has a beautiful integrated cluster support for all the upstream handlers.

Add an *upstream* directive outside the server configuration block:

```
upstream uwsgicluster {
  server unix:///tmp/uwsgi.sock;
  server 192.168.1.235:3031;
  server 10.0.0.17:3017;
}
```

Then modify your uwsgi_pass directive:

```
uwsgi_pass uwsgicluster;
```

Your requests will be balanced between the uWSGI servers configured.

11.8.4 Dynamic apps

The uWSGI server can load applications on demand when passed special vars.

uWSGI can be launched without passing it any application configuration:

```
./uwsgi -s /tmp/uwsgi.sock
```

If a request sets the `UWSGI_SCRIPT` var, the server will load the specified module:

```
location / {
  root html;
  uwsgi_pass uwsgicluster;
  uwsgi_param UWSGI_SCRIPT testapp;
  include uwsgi_params;
}
```

You can even configure multiple apps per-location:

```
location / {
  root html;
  uwsgi_pass uwsgicluster;
  uwsgi_param UWSGI_SCRIPT testapp;
  include uwsgi_params;
}

location /django {
  uwsgi_pass uwsgicluster;
  include uwsgi_params;
  uwsgi_param UWSGI_SCRIPT django_wsgi;
  uwsgi_param SCRIPT_NAME /django;
  uwsgi_modifier1 30;
}
```

The WSGI standard dictates that `SCRIPT_NAME` is the variable used to select a specific application.

The `uwsgi_modifier1 30` option sets the uWSGI modifier `UWSGI_MODIFIER_MANAGE_PATH_INFO`. This per-request modifier instructs the uWSGI server to rewrite the PATH_INFO value removing the SCRIPT_NAME from it.

11.8.5 Static files

For best performance and security, remember to configure Nginx to serve static files instead of letting your poor application handle that instead.

The uWSGI server can serve static files flawlessly but not as quickly and efficiently as a dedicated web server like Nginx.

For example you can the Django `/media` path could be mapped like this:

```
location /media {
  alias /var/lib/python-support/python2.6/django/contrib/admin/media;
}
```

Some applications need to pass control to the UWSGI server only if the requested filename does not exist:

```
if (!-f $request_filename) {
  uwsgi_pass uwsgicluster;
}
```

WARNING

If used incorrectly a configuration like this may cause security problems. For your sanity's sake, double-triple-quadruple check that your application files, configuration files and any other sensitive files are outside of the root of the static files.

11.8.6 Virtual Hosting

You can use Nginx's virtual hosting without particular problems.

If you run "untrusted" web apps (such as those of your clients if you happen to be an ISP) you should limit their memory/address space usage and use a different *uid* for each host/application:

```
server {
  listen 80;
  server_name customersite1.com;
  access_log /var/log/customersite1/access_log;
  location / {
    root /var/www/customersite1;
    uwsgi_pass 127.0.0.1:3031;
    include uwsgi_params;
  }
}

server {
  listen 80;
  server_name customersite2.it;
  access_log /var/log/customersite2/access_log;
  location / {
    root /var/www/customersite2;
    uwsgi_pass 127.0.0.1:3032;
    include uwsgi_params;
  }
}

server {
  listen 80;
  server_name sivusto3.fi;
  access_log /var/log/customersite3/access_log;
  location / {
    root /var/www/customersite3;
    uwsgi_pass 127.0.0.1:3033;
    include uwsgi_params;
  }
}
```

The customers' applications can now be run (using the process manager of your choice, such as *rc.local*, *Running uWSGI via Upstart*, *Supervisord* or whatever strikes your fancy) with a different uid and a limited (if you want) address space for each socket:

```
uwsgi --uid 1001 -w customer1app --limit-as 128 -p 3 -M -s 127.0.0.1:3031
uwsgi --uid 1002 -w customer2app --limit-as 128 -p 3 -M -s 127.0.0.1:3032
uwsgi --uid 1003 -w django3app --limit-as 96 -p 6 -M -s 127.0.0.1:3033
```

CHAPTER 12

Language support

12.1 Python support

12.1.1 The uwsgi Python module

The uWSGI server automagically adds a `uwsgi` module into your Python apps.

This is useful for configuring the uWSGI server, use its internal functions and get statistics (as well as detecting whether you're actually running under uWSGI).

Note: Many of these functions are currently woefully undocumented.

Module-level globals

`uwsgi.`**`numproc`**
: The number of processes/workers currently running.

`uwsgi.`**`buffer_size`**
: The current configured buffer size in bytes.

`uwsgi.`**`started_on`** (*int*)
: The Unix timestamp of uWSGI's startup.

`uwsgi.`**`fastfuncs`**
: This is the dictionary used to define `FastFuncs`.

`uwsgi.`**`applist`**
: This is the list of applications currently configured.

`uwsgi.`**`applications`**
: This is the dynamic applications dictionary.

  See also:

  Application dictionary

`uwsgi.`**`message_manager_marshal`**
: The callable to run when the uWSGI server receives a marshalled message.

`uwsgi.`**`magic_table`**
: The magic table of configuration placeholders.

uwsgi.**opt**
The current configuration options, including any custom placeholders.

Cache functions

uwsgi.**cache_get**(*key*[, *cache_server*])
Get a value from the cache.

Parameters

- **key** – The cache key to read.
- **cache_server** – The UNIX/TCP socket where the cache server is listening. Optional.

uwsgi.**cache_set**(*key*, *value*[, *expire*, *cache_server*])
Set a value in the cache.

Parameters

- **key** – The cache key to write.
- **write** – The cache value to write.
- **expire** – Expiry time of the value, in seconds.
- **cache_server** – The UNIX/TCP socket where the cache server is listening. Optional.

uwsgi.**cache_update**(*key*, *value*[, *expire*, *cache_server*])

uwsgi.**cache_del**(*key*[, *cache_server*])
Delete the given cached value from the cache.

Parameters

- **key** – The cache key to delete.
- **cache_server** – The UNIX/TCP socket where the cache server is listening. Optional.

uwsgi.**cache_exists**(*key*[, *cache_server*])
Quickly check whether there is a value in the cache associated with the given key.

Parameters

- **key** – The cache key to check.
- **cache_server** – The UNIX/TCP socket where the cache server is listening. Optional.

uwsgi.**cache_clear**()

Queue functions

uwsgi.**queue_get**()

uwsgi.**queue_set**()

uwsgi.**queue_last**()

uwsgi.**queue_push**()

uwsgi.**queue_pull**()

uwsgi.**queue_pop**()

uwsgi.**queue_slot**()

uwsgi.**queue_pull_slot**()

SNMP functions

uwsgi.**snmp_set_counter32**(*oidnum*, *value*)

uwsgi.**snmp_set_counter64**(*oidnum*, *value*)

uwsgi.**snmp_set_gauge**(*oidnum*, *value*)

Parameters

- **oidnum** – An integer containing the oid number target.
- **value** – An integer containing the new value of the counter or gauge.

Sets the counter or gauge to a specific value.

uwsgi.**snmp_incr_counter32**(*oidnum*, *value*)

uwsgi.**snmp_incr_counter64**(*oidnum*, *value*)

uwsgi.**snmp_incr_gauge**(*oidnum*, *value*)

uwsgi.**snmp_decr_counter32**(*oidnum*, *value*)

uwsgi.**snmp_decr_counter64**(*oidnum*, *value*)

uwsgi.**snmp_decr_gauge**(*oidnum*, *value*)

Parameters

- **oidnum** – An integer containing the oid number target.
- **value** – An integer containing the amount to increase or decrease the counter or gauge. If not specified the default is 1.

Increases or decreases the counter or gauge by a specific amount.

Note: uWSGI OID tree starts at 1.3.6.1.4.1.35156.17

Spooler functions

uwsgi.**send_to_spooler**(*message_dict=None*, *spooler=None*, *priority=None*, *at=None*, *body=None*, ***kwargs*)

Parameters

- **message_dict** – The message (string keys, string values) to spool. Either this, or **kwargs may be set.
- **spooler** – The spooler (id or directory) to use.
- **priority** – The priority of the message. Larger = less important.
- **at** – The minimum UNIX timestamp at which this message should be processed.
- **body** – A binary (bytestring) body to add to the message, in addition to the message dictionary itself. Its value will be available in the key `body` in the message.

Send data to the *The uWSGI Spooler*. Also known as *spool()*.

Note: Any of the keyword arguments may also be passed in the message dictionary. This means they're reserved words, in a way...

uwsgi.**set_spooler_frequency**(*seconds*)
Set how often the spooler runs.

uwsgi.**spooler_jobs**()

uwsgi.**spooler_pid**()

Advanced methods

uwsgi.**send_message**()
Send a generic message using *The uwsgi Protocol*.

Note: Until version *2f970ce58543278c851ff30e52758fd6d6e69fdc* this function was called `send_uwsgi_message()`.

uwsgi.**route**()

uwsgi.**send_multi_message**()
Send a generic message to multiple recipients using *The uwsgi Protocol*.

Note: Until version *2f970ce58543278c851ff30e52758fd6d6e69fdc* this function was called `send_multi_uwsgi_message()`.

See also:

`Clustering` for examples

uwsgi.**reload**()
Gracefully reload the uWSGI server stack.

See also:

`Reload`

uwsgi.**stop**()

uwsgi.**workers**() → dict
Get a statistics dictionary of all the workers for the current server. A dictionary is returned.

uwsgi.**masterpid**() → int
Return the process identifier (PID) of the uWSGI master process.

uwsgi.**total_requests**() → int
Returns the total number of requests managed so far by the pool of uWSGI workers.

uwsgi.**get_option**()
Also available as *getoption()*.

uwsgi.**set_option**()
Also available as *setoption()*.

uwsgi.**sorry_i_need_to_block**()

uwsgi.**request_id**()

uwsgi.**worker_id**()

uwsgi.**mule_id**()

uwsgi.**log**()

uwsgi.**log_this_request**()

uwsgi.**set_logvar**()

uwsgi.**get_logvar**()

uwsgi.**disconnect**()

uwsgi.**grunt**()

uwsgi.**lock**(*locknum=0*)

Parameters **locknum** – The lock number to lock. Lock 0 is always available.

uwsgi.**is_locked**()

uwsgi.**unlock**(*locknum=0*)

Parameters **locknum** – The lock number to unlock. Lock 0 is always available.

uwsgi.**cl**()

uwsgi.**setprocname**()

uwsgi.**listen_queue**()

uwsgi.**register_signal**(*num*, *who*, *function*)

Parameters

- **num** – the signal number to configure
- **who** – a magic string that will set which process/processes receive the signal.
  - `worker`/`worker0` will send the signal to the first available worker. This is the default if you specify an empty string.
  - `workers` will send the signal to every worker.
  - `workerN` (N > 0) will send the signal to worker N.
  - `mule`/`mule0` will send the signal to the first available mule. (See *uWSGI Mules*)
  - `mules` will send the signal to all mules
  - `muleN` (N > 0) will send the signal to mule N.
  - `cluster` will send the signal to all the nodes in the cluster. Warning: not implemented.
  - `subscribed` will send the signal to all subscribed nodes. Warning: not implemented.
  - `spooler` will send the signal to the spooler.

  `cluster` and `subscribed` are special, as they will send the signal to the master of all cluster/subscribed nodes. The other nodes will have to define a local handler though, to avoid a terrible signal storm loop.
- **function** – A callable that takes a single numeric argument.

uwsgi.**signal**(*num*)

Parameters **num** – the signal number to raise

uwsgi.**signal_wait**([*signum*])

Block the process/thread/async core until a signal is received. Use `signal_received` to get the number of the signal received. If a registered handler handles a signal, `signal_wait` will be interrupted and the actual handler will handle the signal.

Parameters **signum** – Optional - the signal to wait for

uwsgi.**signal_registered**()

uwsgi.**signal_received**()
Get the number of the last signal received. Used in conjunction with `signal_wait`.

uwsgi.**add_file_monitor**()

uwsgi.**add_timer**(*signum*, *seconds*[, *iterations=0*])

Parameters

- **signum** – The signal number to raise.
- **seconds** – The interval at which to raise the signal.
- **iterations** – How many times to raise the signal. 0 (the default) means infinity.

uwsgi.**add_probe**()

uwsgi.**add_rb_timer**(*signum*, *seconds*[, *iterations=0*])
Add an user-space (red-black tree backed) timer.

Parameters

- **signum** – The signal number to raise.
- **seconds** – The interval at which to raise the signal.
- **iterations** – How many times to raise the signal. 0 (the default) means infinity.

uwsgi.**add_cron**(*signal*, *minute*, *hour*, *day*, *month*, *weekday*)
For the time parameters, you may use the syntax `-n` to denote "every n". For instance `hour=-2` would declare the signal to be sent every other hour.

Parameters

- **signal** – The signal number to raise.
- **minute** – The minute on which to run this event.
- **hour** – The hour on which to run this event.
- **day** – The day on which to run this event. This is "OR"ed with `weekday`.
- **month** – The month on which to run this event.
- **weekday** – The weekday on which to run this event. This is "OR"ed with `day`. (In accordance with the POSIX standard, 0 is Sunday, 6 is Monday)

uwsgi.**register_rpc**()

uwsgi.**rpc**()

uwsgi.**rpc_list**()

uwsgi.**call**()

uwsgi.**sendfile**()

uwsgi.**set_warning_message**()

uwsgi.**mem**()

uwsgi.**has_hook**()

uwsgi.**logsize**()

uwsgi.**send_multicast_message**()

uwsgi.**cluster_nodes**()

uwsgi.**cluster_node_name**()

uwsgi.**cluster**()

uwsgi.**cluster_best_node**()

uwsgi.**connect**()

uwsgi.**connection_fd**()

uwsgi.**is_connected**()

uwsgi.**send**()

uwsgi.**recv**()

uwsgi.**recv_block**()

uwsgi.**recv_frame**()

uwsgi.**close**()

uwsgi.**i_am_the_spooler**()

uwsgi.**fcgi**()

uwsgi.**parsefile**()

uwsgi.**embedded_data**(*symbol_name*)

Parameters **string** – The symbol name to extract.

Extracts a symbol from the uWSGI binary image.

See also:

Embedding an application in uWSGI

uwsgi.**extract**()

uwsgi.**mule_msg**(*string*[, *id*])

Parameters

- **string** – The bytestring message to send.
- **id** – Optional - the mule ID to receive the message. If you do not specify an ID, the message will go to the first available programmed mule.

Send a message to a mule.

uwsgi.**farm_msg**()

uwsgi.**mule_get_msg**()

Returns A mule message, once one is received.

Block until a mule message is received and return it. This can be called from multiple threads in the same programmed mule.

uwsgi.**farm_get_msg**()

uwsgi.**in_farm**()

uwsgi.**ready**()

uwsgi.**set_user_harakiri**()

Async functions

uwsgi.**async_sleep**(*seconds*)
Suspend handling the current request for `seconds` seconds and pass control to the next async core.

Parameters **seconds** – Sleep time, in seconds.

uwsgi.**async_connect**()

uwsgi.**async_send_message**()

uwsgi.**green_schedule**()

uwsgi.**suspend**()
Suspend handling the current request and pass control to the next async core clamoring for attention.

uwsgi.**wait_fd_read**(*fd*[, *timeout*])
Suspend handling the current request until there is something to be read on file descriptor `fd`. May be called several times before yielding/suspending to add more file descriptors to the set to be watched.

Parameters

- **fd** – File descriptor number.
- **timeout** – Optional timeout (infinite if omitted).

uwsgi.**wait_fd_write**(*fd*[, *timeout*])
Suspend handling the current request until there is nothing more to be written on file descriptor `fd`. May be called several times to add more file descriptors to the set to be watched.

Parameters

- **fd** – File descriptor number.
- **timeout** – Optional timeout (infinite if omitted).

SharedArea functions

See also:

SharedArea – share memory pages between uWSGI components

uwsgi.**sharedarea_read**(*pos*, *len*) → bytes
Read a byte string from the uWSGI *SharedArea – share memory pages between uWSGI components*.

Parameters

- **pos** – Starting position to read from.
- **len** – Number of bytes to read.

Returns Bytes read, or `None` if the shared area is not enabled or the read request is invalid.

uwsgi.**sharedarea_write**(*pos*, *str*) → long
Write a byte string into the uWSGI *SharedArea – share memory pages between uWSGI components*.

Parameters

- **pos** – Starting position to write to.
- **str** – Bytestring to write.

Returns Number of bytes written, or `None` if the shared area is not enabled or the write could not be fully finished.

uwsgi.**sharedarea_readbyte**(*pos*) → int
Read a single byte from the uWSGI *SharedArea – share memory pages between uWSGI components*.

Parameters **pos** – The position to read from.

Returns Bytes read, or `None` if the shared area is not enabled or the read request is invalid.

uwsgi.**sharedarea_writebyte**(*pos*, *val*) → int
Write a single byte into the uWSGI *SharedArea – share memory pages between uWSGI components*.

Parameters

- **pos** – The position to write the value to.
- **val** (*integer*) – The value to write.

Returns The byte written, or `None` if the shared area is not enabled or the write request is invalid.

uwsgi.**sharedarea_readlong**(*pos*) → int
Read a 64-bit (8-byte) long from the uWSGI *SharedArea – share memory pages between uWSGI components*.

Parameters **pos** – The position to read from.

Returns The value read, or `None` if the shared area is not enabled or the read request is invalid.

uwsgi.**sharedarea_writelong**(*pos*, *val*) → int
Write a 64-bit (8-byte) long into the uWSGI *SharedArea – share memory pages between uWSGI components*.

Parameters

- **pos** – The position to write the value to.
- **val** (*long*) – The value to write.

Returns The value written, or `None` if the shared area is not enabled or the write request is invalid.

uwsgi.**sharedarea_inclong**(*pos*) → int
Atomically increment a 64-bit long value in the uWSGI *SharedArea – share memory pages between uWSGI components*.

Parameters **pos** – The position of the value.

Returns The new value at the given position, or `None` if the shared area is not enabled or the read request is invalid.

Erlang functions

uwsgi.**erlang_send_message**(*node*, *process_name*, *message*)

uwsgi.**erlang_register_process**(*process_name*, *callable*)

uwsgi.**erlang_recv_message**(*node*)

uwsgi.**erlang_connect**(*address*)

Returns File descriptor or -1 on error

uwsgi.**erlang_rpc**(*node*, *module*, *function*, *argument*)

12.1.2 uWSGI API - Python decorators

The uWSGI API is very low-level, as it must be language-independent.

That said, being too low-level is not a Good Thing for many languages, such as Python.

Decorators are, in our humble opinion, one of the more kick-ass features of Python, so in the uWSGI source tree you will find a module exporting a bunch of decorators that cover a good part of the uWSGI API.

Notes

Signal-based decorators execute the signal handler in the first available worker. If you have enabled the spooler you can execute the signal handlers in it, leaving workers free to manage normal requests. Simply pass `target='spooler'` to the decorator.

```
@timer(3, target='spooler')
def hello(signum):
    print("hello")
```

Example: a Django session cleaner and video encoder

Let's define a `task.py` module and put it in the Django project directory.

```
from uwsgidecorators import *
from django.contrib.sessions.models import Session
import os

@cron(40, 2, -1, -1, -1)
def clear_django_session(num):
    print("it's 2:40 in the morning: clearing django sessions")
    Session.objects.all().delete()

@spool
def encode_video(arguments):
    os.system("ffmpeg -i \"%s\" image%%d.jpg" % arguments['filename'])
```

The session cleaner will be executed every day at 2:40, to enqueue a video encoding we simply need to spool it from somewhere else.

```
from task import encode_video

def index(request):
    # launching video encoding
    encode_video.spool(filename=request.POST['video_filename'])
    return render_to_response('enqueued.html')
```

Now run uWSGI with the spooler enabled:

```
[uwsgi]
; a couple of placeholder
django_projects_dir = /var/www/apps
my_project = foobar
; chdir to app project dir and set pythonpath
chdir = %(django_projects_dir)/%(my_project)
pythonpath = %(django_projects_dir)
; load django
module = django.core.handlers:WSGIHandler()
env = DJANGO_SETTINGS_MODULE=%(my_project).settings
; enable master
master = true
; 4 processes should be enough
processes = 4
; enable the spooler (the mytasks dir must exist!)
```

```
spooler = %(chdir)/mytasks
; load the task.py module
import = task
; bind on a tcp socket
socket = 127.0.0.1:3031
```

The only especially relevant option is the `import` one. It works in the same way as `module` but skips the WSGI callable search. You can use it to preload modules before the loading of WSGI apps. You can specify an unlimited number of ‘‘‘import’‘ directives.

Example: web2py + spooler + timer

First of all define your spooler and timer functions (we will call it :file:`mytasks.py`)

```
from uwsgidecorators import *

@spool
def a_long_task(args):
    print(args)

@spool
def a_longer_task(args)
    print("longer.....")

@timer(3)
def three_seconds(signum):
    print("3 seconds elapsed")

@timer(10, target='spooler')
def ten_seconds_in_the_spooler(signum):
    print("10 seconds elapsed in the spooler")
```

Now run web2py.

```
uwsgi --socket :3031 --spooler myspool --master --processes 4 --import mytasks --module web2py.wsgiha
```

As soon as the application is loaded, you will see the 2 timers running in your logs.

Now we want to enqueue tasks from our web2py controllers.

Edit one of them and add

```
import mytasks # be sure mytasks is importable!

def index(): # this is a web2py action
    mytasks.a_long_task.spool(foo='bar')
    return "Task enqueued"
```

uwsgidecorators API reference

uwsgidecorators.**postfork**(*func*)

uWSGI is a preforking (or “fork-abusing”) server, so you might need to execute a fixup task after each `fork()`. The `postfork` decorator is just the ticket. You can declare multiple `postfork` tasks. Each decorated function will be executed in sequence after each `fork()`.

```
@postfork
def reconnect_to_db():
```

```
        myfoodb.connect()

    @postfork
    def hello_world():
        print("Hello World")
```

uwsgidecorators.**spool**(*func*)

The uWSGI *spooler* can be very useful. Compared to Celery or other queues it is very "raw". The `spool` decorator will help!

```
@spool
def a_long_long_task(arguments):
    print(arguments)
    for i in xrange(0, 10000000):
        time.sleep(0.1)

@spool
def a_longer_task(args):
    print(args)
    for i in xrange(0, 10000000):
        time.sleep(0.5)

# enqueue the tasks
a_long_long_task.spool(foo='bar',hello='world')
a_longer_task.spool({'pippo':'pluto'})
```

The functions will automatically return `uwsgi.SPOOL_OK` so they will be executed one time independently by their return status.

uwsgidecorators.**spoolforever**(*func*)

Use `spoolforever` when you want to continuously execute a spool task. A `@spoolforever` task will always return `uwsgi.SPOOL_RETRY`.

```
@spoolforever
def a_longer_task(args):
    print(args)
    for i in xrange(0, 10000000):
        time.sleep(0.5)

# enqueue the task
a_longer_task.spool({'pippo':'pluto'})
```

uwsgidecorators.**spoolraw**(*func*)

Advanced users may want to control the return value of a task.

```
    @spoolraw
    def a_controlled_task(args):
        if args['foo'] == 'bar':
            return uwsgi.SPOOL_OK
        return uwsgi.SPOOL_RETRY

    a_controlled_task.spool(foo='bar')
```

uwsgidecorators.**rpc**(*"name"*, *func*)

uWSGI *uWSGI RPC Stack* is the fastest way to remotely call functions in applications hosted in uWSGI instances. You can easily define exported functions with the @rpc decorator.

```
@rpc('helloworld')
def ciao_mondo_function():
```

```
        return "Hello World"
```

uwsgidecorators.**signal**(*num)(func*)
You can register signals for the *signal framework* in one shot.

```
@signal(17)
def my_signal(num):
    print("i am signal %d" % num)
```

uwsgidecorators.**timer**(*interval*, *func*)
Execute a function at regular intervals.

```
@timer(3)
def three_seconds(num):
    print("3 seconds elapsed")
```

uwsgidecorators.**rbtimer**(*interval*, *func*)
Works like @timer but using red black timers.

uwsgidecorators.**cron**(*min*, *hour*, *day*, *mon*, *wday*, *func*)
Easily register functions for the `CronInterface`.

```
@cron(59, 3, -1, -1, -1)
def execute_me_at_three_and_fiftynine(num):
    print("it's 3:59 in the morning")
```

Since 1.2, a new syntax is supported to simulate `crontab`-like intervals (every Nth minute, etc.). `*/5 * * * *` can be specified in uWSGI like thus:

```
@cron(-5, -1, -1, -1, -1)
def execute_me_every_five_min(num):
    print("5 minutes, what a long time!")
```

uwsgidecorators.**filemon**(*path*, *func*)
Execute a function every time a file/directory is modified.

```
@filemon("/tmp")
def tmp_has_been_modified(num):
    print("/tmp directory has been modified. Great magic is afoot")
```

uwsgidecorators.**erlang**(*process_name*, *func*)
Map a function as an *Erlang* process.

```
@erlang('foobar')
def hello():
    return "Hello"
```

uwsgidecorators.**thread**(*func*)
Mark function to be executed in a separate thread.

Important: Threading must be enabled in uWSGI with the `enable-threads` or `threads <n>` option.

```
@thread
def a_running_thread():
    while True:
        time.sleep(2)
        print("i am a no-args thread")

@thread
```

```
def a_running_thread_with_args(who):
    while True:
        time.sleep(2)
        print("Hello %s (from arged-thread)" % who)

a_running_thread()
a_running_thread_with_args("uWSGI")
```

You may also combine `@thread` with `@postfork` to spawn the postfork handler in a new thread in the freshly spawned worker.

```
@postfork
@thread
def a_post_fork_thread():
    while True:
        time.sleep(3)
        print("Hello from a thread in worker %d" % uwsgi.worker_id())
```

uwsgidecorators.**lock**(*func*)

This decorator will execute a function in fully locked environment, making it impossible for other workers or threads (or the master, if you're foolish or brave enough) to run it simultaneously. Obviously this may be combined with @postfork.

```
@lock
def dangerous_op():
    print("Concurrency is for fools!")
```

uwsgidecorators.**mulefunc**([*mulespec*], *func*)

Offload the execution of the function to a *mule*. When the offloaded function is called, it will return immediately and execution is delegated to a mule.

```
@mulefunc
def i_am_an_offloaded_function(argument1, argument2):
    print argument1,argument2
```

You may also specify a mule ID or mule farm to run the function on. Please remember to register your function with a uwsgi import configuration option.

```
@mulefunc(3)
def on_three():
    print "I'm running on mule 3."

@mulefunc('old_mcdonalds_farm')
def on_mcd():
    print "I'm running on a mule on Old McDonalds' farm."
```

uwsgidecorators.**harakiri**(*time*, *func*)

Starting from uWSGI 1.3-dev, a customizable secondary *harakiri* subsystem has been added. You can use this decorator to kill a worker if the given call is taking too long.

```
@harakiri(10)
def slow_function(foo, bar):
    for i in range(0, 10000):
        for y in range(0, 10000):
            pass

# or the alternative lower level api

uwsgi.set_user_harakiri(30) # you have 30 seconds. fight!
```

```
    slow_func()
    uwsgi.set_user_harakiri(0) # clear the timer, all is well
```

12.1.3 Pump support

Note: Pump is not a PEP nor a standard.

Pump is a new project aiming at a "better" WSGI.

An example Pump app, for your convenience:

```
def app(req):
    return {
        "status": 200,
        "headers": {"content_type": "text/html"},
        "body": "<h1>Hello!</h1>"
    }
```

To load a Pump app simply use the `pump` option to declare the callable.

```
uwsgi --http-socket :8080 -M -p 4 --pump myapp:app
```

`myapp` is the name of the module (that must be importable!) and app is the callable. The callable part is optional – by default uWSGI will search for a callable named 'application'.

12.1.4 Python Tracebacker

New in version 1.3-dev.

Usually if you want to get a real-time traceback from your app you'd have to modify your code to add a hook or entry point for that as described on the `TipsAndTricks` page.

Starting from 1.3-dev, uWSGI includes a similar technique allowing you to get realtime traceback via a UNIX socket.

To enable the tracebacker, add the option `py-tracebacker=<socket>` where `<socket>` is the _basename_ for the created UNIX sockets.

If you have 4 uWSGI workers and you add `py-tracebacker=/tmp/tbsocket`, four sockets named `/tmp/tbsocket1` through `/tmp/tbsocket4` will be created.

Connecting to one of them will return the current traceback of the threads running in the worker. To connect to those sockets you can use whatever application or method you like the best, but uWSGI includes a convenience option `connect-and-read` you can use:

```
uwsgi --connect-and-read /tmp/tbsocket1
```

An example

Let's write a silly test application called `slow.py`:

```
import time

def dormi():
  time.sleep(60)

def dormi2():
```

```
  dormi()

def dormi3():
  dormi2()

def dormi4():
  dormi3()

def dormi5():
  dormi4()

def application(e, start_response):
  start_response('200 OK', [('Content-Type', 'text/html')])
  dormi5()
  return "hello"
```

And then run it:

```
uwsgi --http :8080 -w slow --master --processes 2 --threads 4 --py-tracebacker /tmp/tbsocket.
```

Then make a bunch of requests into it:

```
curl http://localhost:8080 &
curl http://localhost:8080 &
curl http://localhost:8080 &
curl http://localhost:8080 &
```

Now, while these requests are running (they'll take pretty much exactly a minute to complete each), you can retrieve the traceback for, let's say, the two first workers:

```
./uwsgi --connect-and-read /tmp/tbsocket.1
./uwsgi --connect-and-read /tmp/tbsocket.2
```

The tracebacker output will be something like this:

```
*** uWSGI Python tracebacker output ***

thread_id = uWSGIWorker1Core1 filename = ./slow.py lineno = 22 function = application line = dormi5()
thread_id = uWSGIWorker1Core1 filename = ./slow.py lineno = 14 function = dormi5 line = def dormi5():
thread_id = uWSGIWorker1Core1 filename = ./slow.py lineno = 13 function = dormi4 line = def dormi4():
thread_id = uWSGIWorker1Core1 filename = ./slow.py lineno = 12 function = dormi3 line = def dormi3():
thread_id = uWSGIWorker1Core1 filename = ./slow.py lineno = 11 function = dormi2 line = def dormi2():
thread_id = uWSGIWorker1Core1 filename = ./slow.py lineno = 9 function = dormi line = time.sleep(60)

thread_id = uWSGIWorker1Core3 filename = ./slow.py lineno = 22 function = application line = dormi5()
thread_id = uWSGIWorker1Core3 filename = ./slow.py lineno = 14 function = dormi5 line = def dormi5():
thread_id = uWSGIWorker1Core3 filename = ./slow.py lineno = 13 function = dormi4 line = def dormi4():
thread_id = uWSGIWorker1Core3 filename = ./slow.py lineno = 12 function = dormi3 line = def dormi3():
thread_id = uWSGIWorker1Core3 filename = ./slow.py lineno = 11 function = dormi2 line = def dormi2():
thread_id = uWSGIWorker1Core3 filename = ./slow.py lineno = 9 function = dormi line = time.sleep(60)

thread_id = MainThread filename = ./slow.py lineno = 22 function = application line = dormi5()
thread_id = MainThread filename = ./slow.py lineno = 14 function = dormi5 line = def dormi5(): dormi4
thread_id = MainThread filename = ./slow.py lineno = 13 function = dormi4 line = def dormi4(): dormi3
thread_id = MainThread filename = ./slow.py lineno = 12 function = dormi3 line = def dormi3(): dormi2
thread_id = MainThread filename = ./slow.py lineno = 11 function = dormi2 line = def dormi2(): dormi
thread_id = MainThread filename = ./slow.py lineno = 9 function = dormi line = time.sleep(60)
```

Combining the tracebacker with Harakiri

If a request is killed by the *harakiri* feature, a traceback is automatically logged during the Harakiri phase.

12.1.5 Aliasing Python modules

Having multiple version of a Python package/module/file is very common.

Manipulating PYTHONPATH or using virtualenvs are a way to use various versions without changing your code.

But hey, why not have an aliasing system that lets you arbitrarily map module names to files? That's why we have the `pymodule-alias` option!

Case 1 - Mapping a simple file to a virtual module

Let's say we have `swissknife.py` that contains lots of useful classes and functions.

It's imported in gazillions of places in your app. Now, we'll want to modify it, but keep the original file intact for whichever reason, and call it `swissknife_mk2`.

Your options would be

1. to modify all of your code to import and use swissknife_mk2 instead of swissknife. Yeah, no, not's going to happen.
2. modify the first line of all your files to read `import swissknife_mk2 as swissknife`. A lot better but you make software for money... and time is money, so why the fuck not use something more powerful?

So don't touch your files – just remap!

```
./uwsgi -s :3031 -w myproject --pymodule-alias swissknife=swissknife_mk2
# Kapow! uWSGI one-two ninja punch right there!
# You can put the module wherever you like, too:
./uwsgi -s :3031 -w myproject --pymodule-alias swissknife=/mnt/floppy/KNIFEFAC/SWISSK~1.PY
# Or hey, why not use HTTP?
./uwsgi -s :3031 -w myproject --pymodule-alias swissknife=http://uwsgi.it/modules/swissknife_extreme
```

You can specify multiple `pymodule-alias` directives.

```
uwsgi:
  socket: :3031
  module: myproject
  pymodule-alias: funnymodule=/opt/foo/experimentalfunnymodule.py
  pymodule-alias: uglymodule=/opt/foo/experimentaluglymodule.py
```

Case 2 - mapping a packages to directories

You have this shiny, beautiful Django project and something occurs to you: Would it work with Django trunk? On to set up a new virtualenv... nah. Let's just use `pymodule-alias`!

```
./uwsgi -s :3031 -w django_uwsgi --pymodule-alias django=django-trunk/django
```

Case 3 - override specific submodules

You have a Werkzeug project where you want to override - for whichever reason - `werkzeug.test_app` with one of your own devising. Easy, of course!

```
./uwsgi -s :3031 -w werkzeug.testapp:test_app() --pymodule-alias werkzeug.testapp=mytestapp
```

See also:

Python configuration options

12.1.6 Application dictionary

You can use the application dictionary mechanism to avoid setting up your application in your configuration.

```
import uwsgi
import django.core.handlers.wsgi

application = django.core.handlers.wsgi.WSGIHandler()

def myapp(environ, start_response):
  start_response('200 OK', [('Content-Type', 'text/plain')])
  yield 'Hello World\n'

uwsgi.applications = {
  '': application,
  '/django': 'application',
  '/myapp': myapp
}
```

Passing this Python module name (that is, it should be importable and without the `.py` extension) to uWSGI's `module` / `wsgi` option, uWSGI will search the `uwsgi.applications` dictionary for the URL prefix/callable mappings.

The value of every item can be a callable, or its name as a string.

12.1.7 Virtualenv support

virtualenv is a mechanism that lets you isolate one (or more) Python applications' libraries (and interpreters, when not using uWSGI) from each other. Virtualenvs should be used by any respectable modern Python application.

Quickstart

1. Create your virtualenv:

   ```
   $ virtualenv myenv
   New python executable in myenv/bin/python
   Installing setuptools..............done.
   Installing pip.........done.
   ```

2. Install all the modules you need (using Flask as an example):

   ```
   $ ./myenv/bin/pip install flask
   $ # Many modern Python projects ship with a `requirements.txt` file that you can use with pip li
   $ ./myenv/bin/pip install -r requirements.txt
   ```

3. Copy your WSGI module into this new environment (under `lib/python2.x` if you do not want to modify your `PYTHONPATH`).

   Note: It's common for many deployments that your application will live outside the virtualenv. How to configure this is not quite documented yet, but it's probably very easy.

Run the uwsgi server using the `home`/`virtualenv` option (`-H` for short):

```
$ uwsgi -H myenv -s 127.0.0.1:3031 -M -w envapp
```

12.1.8 Python 3

Python 3 got updated WSGI spec PEP3333. It requires applications to respond with `bytes`-instances, not strings, back to WSGI stack.

You should encode strings or use bytes literals:

```
def application(environ, start_response):
  start_response('200 OK', [('Content-Type', 'text/plain')])
  yield 'Hello '.encode('utf-8')
  yield b'World\n'
```

12.1.9 Paste support

If you are a user or developer of Paste-compatible frameworks, such as Pylons and Turbogears or applications using them, you can use the uWSGI `--paste` option to conveniently deploy your application.

For example, if you have a virtualenv in `/opt/tg2env` containing a Turbogears app called `addressbook` configured in `/opt/tg2env/addressbook/development.ini`:

```
uwsgi --paste config:/opt/tg2env/addressbook/development.ini --socket :3031 -H /opt/tg2env
```

That's it! No additional configuration or Python modules to write.

> **Warning:** If you setup multiple process/workers (*master* mode) you will receive an error:
>
> ```
> AssertionError: The EvalException middleware is not usable in a multi-process environment
> ```
>
> in which case you'll have to set the `debug` option in your paste configuration file to False – or revert to single process environment.

12.1.10 Pecan support

If you are a user or developer of the Pecan WSGI framework, you can use the uWSGI `--pecan` option to conveniently deploy your application.

For example, if you have a virtualenv in `/opt/pecanenv` containing a Pecan app called `addressbook` configured in `/opt/pecanenv/addressbook/development.py`:

```
uwsgi --pecan /opt/pecanenv/addressbook/development.py --socket :3031 -H /opt/pecanenv
```

> **Warning:** If you setup multiple process/workers (*master* mode) you will receive an error:
>
> ```
> AssertionError: The DebugMiddleware middleware is not usable in a multi-process environment
> ```
>
> in which case you'll have to set the `debug` option in your pecan configuration file to False – or revert to single process environment.

12.1.11 Using the uwsgi_admin Django app

First of all you need to get the `uwsgi_admin` app from https://github.com/unbit/uwsgi_django (once it was in the `django` directory of the distribution).

It plugs into Django's admin app, so if `uwsgi_admin` is importable, just add it into your `INSTALLED_APPS`.

```
INSTALLED_APPS = (
    # ...
    'django.contrib.admin',
    'uwsgi_admin',
    # ...
)
```

Then modify your `urls.py` accordingly. For example:

```
# ...
url(r'^admin/uwsgi/', include('mysite.uwsgi_admin.urls')),
url(r'^admin/', include(admin.site.urls)),
# ...
```

Be sure to place the URL pattern for uwsgi_admin *before* the one for the admin site, or it will never match.

`/admin/uwsgi/` will then serve uWSGI statistics and has a button for graceful reloading of the server (when running under a Master). Note that memory usage is reported only when the `memory-report` option is enabled.

12.2 The PyPy plugin

12.2.1 Benchmarks for the PyPy plugin

Note: this benchmark is now (november 2013) very outdated, most of the numbers here have changed (in better) with newer PyPy releases

---

This is mainly targeted at PyPy developers to spot slow paths or to fix corner-case bugs.

uWSGI stresses lot of areas of PyPy (most of them rarely used in pure-Python apps), so making these benchmarks is good both for uWSGI and PyPy.

- Results are rounded for ease of reading. Each test is executed 10 times on an 8-core Intel i7-3615QM CPU @ 2.30GHz.
- The CPython version is 2.7.5, PyPy is latest tip at 2013-05-23.
- Tests are run with logging disabled.
- Tests are run without thunder locking
- The client suite introduces ad-hoc errors and disconnections, so numbers are way lower that what you can get with 'ab' or 'httperf'

Generally the command lines are:

```
uwsgi --http-socket :9090 --wsgi hello --disable-logging
uwsgi --http-socket :9090 --pypy-home /opt/pypy --pypy-wsgi hello --disable-logging
```

Simple Hello World

The most useless of the tests (as it shows only how uWSGI performs instead of the chosen Python engine).

```
def application(e, sr):
    sr('200 Ok', [('Content-Type', 'text/html')])
    return "ciao"
```

CPython: 6500 RPS, memory used 7MB (no leak detected)

Syscalls used:

```
0.000403 gettimeofday({1369293059, 218207}, NULL) = 0
0.000405 read(5, "GET / HTTP/1.1\r\nUser-Agent: curl/7.30.0\r\nHost: ubuntu64.local:9090\r\nAccept: *
0.000638 write(5, "HTTP/1.1 200 Ok\r\nContent-Type: text/html\r\n\r\n", 44) = 44
0.000678 write(5, "ciao", 4)        = 4
0.000528 gettimeofday({1369293059, 220477}, NULL) = 0
0.000394 close(5)
```

PyPy: 6560 RPS, memory used 71MB (no leak detected)

Syscalls: No differences with CPython

Considerations:

- There is only slightly (read: irrelevant) better performance in PyPy.
- Memory usage is 10x higher with PyPy. This is caused by the difference in the binary size (about 4 megs for libpython, about 50 for stripped libpypy-c). It is important to note that this 10x increase is only on startup, after the app is loaded memory allocations are really different. It looks like the PyPy team is working on reducing the binary size too.

CPU bound test (fibonacci)

```
def fib(n):
   if n == 0:
       return 0
   if n == 1:
       return 1
   return fib(n-1) + fib(n-2)

def application(e, sr):
   sr('200 Ok', [('Content-Type', 'text/html')])
   fib(36)
   return "ciao"
```

This is where PyPy shines.

- CPython: time-to-complete 6400 milliseconds, memory used 65 MB
- PyPy: time-to-complete 900 milliseconds, memory used 71 MB
- The response time here is astonishing, there is no debate about how much better PyPy can be for CPU intensive (and/or highly recursive) tasks.
- More interesting is how the memory usage of PyPy remains the same of the simple hello world, while CPython's increased tenfold.
- Syscall usage is again the same.

Werkzeug testapp

You may think this is not very different from the Hello World example, but this specific application does actually call lots of Python functions and inspects the entire WSGI `environ` dictionary. This is very near to a standard application without I/O.

CPython: 600 RPS, memory usage 13MB

Syscalls:

```
0.000363 gettimeofday({1369294531, 360307}, NULL) = 0
0.000421 read(5, "GET / HTTP/1.1\r\nUser-Agent: curl/7.30.0\r\nHost: ubuntu64.local:9090\r\nAccept: *
0.002046 getcwd("/root/uwsgi", 1024) = 12
0.000483 stat("/root/uwsgi/.", {st_mode=S_IFDIR|0755, st_size=12288, ...}) = 0
0.000602 stat("/usr/local/lib/python2.7/dist-packages/greenlet-0.4.0-py2.7-linux-x86_64.egg", {st_mod
0.000530 stat("/usr/local/lib/python2.7/dist-packages/gevent-1.0dev-py2.7-linux-x86_64.egg", {st_mode
0.000506 stat("/usr/lib/python2.7", {st_mode=S_IFDIR|0755, st_size=28672, ...}) = 0
0.000440 stat("/usr/lib/python2.7/plat-x86_64-linux-gnu", {st_mode=S_IFDIR|0755, st_size=4096, ...})
0.000463 stat("/usr/lib/python2.7/lib-tk", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
0.000624 stat("/usr/lib/python2.7/lib-old", 0x7fffb70da6a0) = -1 ENOENT (No such file or directory)
0.000434 stat("/usr/lib/python2.7/lib-dynload", {st_mode=S_IFDIR|0755, st_size=12288, ...}) = 0
0.000515 stat("/usr/local/lib/python2.7/dist-packages", {st_mode=S_IFDIR|S_ISGID|0775, st_size=4096,
0.000569 stat("/usr/lib/python2.7/dist-packages", {st_mode=S_IFDIR|0755, st_size=12288, ...}) = 0
0.000387 stat("/usr/lib/python2.7/dist-packages/gtk-2.0", {st_mode=S_IFDIR|0755, st_size=4096, ...})
0.000347 stat("/usr/lib/pymodules/python2.7", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
0.000675 write(5, "HTTP/1.1 200 OK\r\nContent-Type: text/html; charset=utf-8\r\nContent-Length: 7554\
0.000575 write(5, "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\"\n  \"http://www.w
0.000469 gettimeofday({1369294531, 370471}, NULL) = 0
0.000391 close(5)                       = 0
```

PyPy: 1500 RPSs, memory usage 74MB

Syscalls:

```
0.000397 gettimeofday({1369294713, 743204}, NULL) = 0
0.000431 read(5, "GET / HTTP/1.1\r\nUser-Agent: curl/7.30.0\r\nHost: ubuntu64.local:9090\r\nAccept: *
0.003217 gettimeofday({1369294713, 746909}, NULL) = 0
0.000660 gettimeofday({1369294713, 747509}, NULL) = 0
0.000958 gettimeofday({1369294713, 748463}, NULL) = 0
0.000359 gettimeofday({1369294713, 748832}, NULL) = 0
0.000586 gettimeofday({1369294713, 749427}, NULL) = 0
0.000660 gettimeofday({1369294713, 750077}, NULL) = 0
0.000626 gettimeofday({1369294713, 750695}, NULL) = 0
0.000318 gettimeofday({1369294713, 751010}, NULL) = 0
0.000598 gettimeofday({1369294713, 751586}, NULL) = 0
0.000782 gettimeofday({1369294713, 752391}, NULL) = 0
0.000738 gettimeofday({1369294713, 753129}, NULL) = 0
0.000355 gettimeofday({1369294713, 753483}, NULL) = 0
0.000617 gettimeofday({1369294713, 754156}, NULL) = 0
0.000502 gettimeofday({1369294713, 754649}, NULL) = 0
0.000484 gettimeofday({1369294713, 755139}, NULL) = 0
0.000513 gettimeofday({1369294713, 755674}, NULL) = 0
0.001537 getcwd("/opt/uwsgi", 256) = 12
0.000641 stat("/opt/uwsgi/.", {st_mode=S_IFDIR|0755, st_size=12288, ...}) = 0
0.000668 stat("/opt/pypy/site-packages/setuptools-0.6c11-py2.7.egg", {st_mode=S_IFREG|0644, st_size=3
0.000766 stat("/opt/pypy/site-packages/pip-1.3.1-py2.7.egg", {st_mode=S_IFDIR|0755, st_size=4096, ...
0.000652 stat("/opt/pypy/lib_pypy/__extensions__", 0x7ff66a446030) = -1 ENOENT (No such file or direc
0.000570 stat("/opt/pypy/lib_pypy", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
0.000688 stat("/opt/pypy/lib-python/2.7", {st_mode=S_IFDIR|0755, st_size=12288, ...}) = 0
0.000592 stat("/opt/pypy/lib-python/2.7/lib-tk", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
```

```
0.000534 stat("/opt/pypy/lib-python/2.7/plat-linux2", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
0.000840 stat("/opt/pypy/site-packages", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
0.000592 stat("/opt/uwsgi/.", {st_mode=S_IFDIR|0755, st_size=12288, ...}) = 0
0.001014 write(5, "HTTP/1.1 200 OK\r\nContent-Type: text/html; charset=utf-8\r\nContent-Length: 4915\
0.000510 write(5, "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\"\n  \"http://www.w
0.000729 gettimeofday({1369294713, 766079}, NULL) = 0
0.000616 close(5)                       = 0
```

Considerations:

- This test stresses standard function calls. We have about 2.5x improvement with PyPy, while memory usage is pretty similar (considering the 62MB base difference).
- There is a syscall "problem" with PyPy, soon before starting the path checks it calls a blast of `gettimeofday()` syscalls. Without these, the RPS could increase a bit.

Werkzeug testapp with multithreading

It looks like Armin Rigo will soon be able to remove the GIL from PyPy. While he works on this, we can compare multithreading CPython and PyPy.

Multithreading for Python webapps is a good approach, the GIL is generally released during the blocking parts, so you can safely punch the face of people ranting on the slowness of Python threads without knowing the context.

We spawn 8 threads (with Linux default stack size), and we stress test with a concurrency of 10.

- CPython: 200 RPSs, memory usage 14MB
- PyPy: 1100 RPSs, memory usage 88 MB

Here we have a problem. To avoid the possibility of a uWSGI threading bug we added a comparative test with `mod_wsgi` in embedded mode (as uWSGI's threading model is based on `mod_wsgi`). Results are the same (between 160 and 190 in apache2+mod_wsgi). So it looks like multithreading in PyPy is way better.

We cannot, however, exclude other problems (testing threads is really hard).

Memory usage is a bit higher on PyPy (about 1.5 megs per thread compared to less than 200k in cpython)

Syscalls report will be hard to print, but the same blast of `gettimeofday` can be noted on PyPy, while lock contention seems the same between uWSGI/mod_wsgi and PyPy.

RPC

uWSGI RPC is good for testing string manipulation. RPC parsing is done in C with the CPython plugin and in Python in PyPy. RPC is called using the internal routing system (as the PyPy plugin does not export the `uwsgi.rpc()` API function yet).

The option added to both command lines is:

```
--route-run "rpc myfunc:one two threee four five six seven"
```

while the function is registered as:

```
import uwsgi

def myfunc(*args):
    return '|'.join(reversed(args))

uwsgi.register_rpc('myfunc', myfunc)
```

The results are pretty similar to the "hello world" one.

- CPython: 6400 RPSs, 8MB memory usage
- PyPy: 6500 RPSs, 71MB memory usage

PyPy has a small, "irrelevant" advantage in term of performance, but do remember its string parsing is done in pure Python.

RPC (multithread)

Here we have very interesting results:

- CPython: 6300 RPSs, 8MB memory usage
- PyPy: 6000 RPSs, 71MB memory usage

This time it is easy to understand what is going on. In PyPy the GIL is held 99% of the time in RPC mode (as message parsing is done in Python), while the CPython version we have the GIL only for 10% of the whole request time.

Rewriting the RPC parsing in `cffi` will probably change the results to look more like the Werkzeug numbers. Something to look at in the future, unless Armin manages to remove the GIL.

Notes

- Testing multiprocessing is useless, do not ask for it.
- The uWSGI PyPy plugin still does not support all of the features of the CPython based plugin, we cannot exclude a little drop in performance while we add features.
- These numbers might look low to you if you have already made (or read) benchmarks. This is because the test tool injects bad requests in the stream to test server robustness.
- Again, this tests are only useful for the PyPy and uWSGI teams, do not base your choice between CPython and PyPy on them! (Your app's requirements will always be unique, and it's very possible that your app won't even run on PyPy even though it chugs along fine on CPython.)

12.2.2 Introduction

Idea/Design: Maciej Fijalkowski

Contributors: Alex Gaynor, Armin Rigo

A new PyPy plugin based on cffi is available since uWSGI 1.9.11. The old slow cpyext-based one has been removed from the tree.

The plugin is currently supported only on Linux systems. Following releases will support other platforms as well.

The plugin loads `libpypy-s.so` on startup, sets the home of the PyPy installation and executes a special Python module implementing the plugin logic. So yes, most of the plugin is implemented in Python, and theoretically this approach will allow writing uWSGI plugins directly in Python in addition to C, C++ and Objective-C.

As of August 2013 all of the required patches to PyPy have been merged, so you can get an official source tarball release. Unfortunately you still need to build/translate libpypy-c by yourself, or download one of the following files (they require libssl 1.0):

- Linux x86 32-bit: http://projects.unbit.it/downloads/pypy/libpypy-c-x86_32_20130524.so
- Linux x86 32-bit (with debug symbols): http://projects.unbit.it/downloads/pypy/libpypy-c-x86_32_20130524-dbg.so

- Linux x86 64-bit: http://projects.unbit.it/downloads/pypy/libpypy-c-x86_64_20130524.so
- Linux x86 64-bit (with debug symbols): http://projects.unbit.it/downloads/pypy/libpypy-c-x86_64_20130524-dbg.so

In addition to the library you need to obviously download an official binary tarball too.

UPDATE:

As of August 2013, libpypy-c is not automatically build in pypy release tarballs or nightly builds. If you do not want to translate the whole tree every time, you can try the "alternative approach" (described below)

12.2.3 Building libpypy-c (if needed)

Get PyPy sources and translate it. This should require no more than 50 minutes, but be sure to have at least 2 gigabytes of free memory on 32-bit systems and 4 gigabytes for 64-bit systems.

```
./rpython/bin/rpython -Ojit --shared --gcrootfinder=shadowstack pypy/goal/targetpypystandalone
```

12.2.4 Install uWSGI with PyPy support

As always with uWSGI, you have different ways to install uWSGI based on your needs.

If you have installed pip in your PyPy home, you can run

```
pip install uwsgi
```

The uwsgi setup.py file will recognize the PyPy environment and will build a PyPy-only uWSGI binary.

You can compile manually:

```
UWSGI_PROFILE=pypy make
```

Or you can use the network installer:

```
curl http://uwsgi.it/install | bash -s pypy /tmp/uwsgi
```

This will build a uWSGI + PyPy binary in `/tmp/uwsgi`.

Or you can build PyPy support as a plugin.

```
python uwsgiconfig.py --plugin plugins/pypy
```

12.2.5 The PyPy home

The uWSGI Python plugin (more exactly the CPython plugin) works by linking in `libpython`. That means you need to rebuild the plugin for every different version of Python. The PyPy plugin is different, as libpypy-c is loaded on startup and its symbols are resolved at runtime. This allows you to migrate to a different PyPy version on the fly.

The downside of this approach is that you need to inform uWSGI where your PyPy installation is at runtime.

Supposing your PyPy is in `/opt/pypy` you can start uWSGI with:

```
uwsgi --http-socket :9090 --pypy-home /opt/pypy
```

With this command line uWSGI will search for `/opt/pypy/libpypy-c.so` and if found, it will set that path as the PyPy home.

If your `libpypy-c.so` is outside of the PyPy home (and in a directory not reachable by the dynamic linker), you can use the ‘‘–pypy-lib‘‘option.

```
uwsgi --http-socket :9090 --pypy-home /opt/pypy --pypy-lib /opt/libs/libpypy-c.so
```

With this approach you are able to use the library from a specific PyPy build and the home from another one.

Note: Remember to prefix –pypy-lib with ./ if you want to point to a .so file in your current directory!

12.2.6 The PyPy setup file

As said before, most of the uWSGI PyPy plugin is written in Python. This code is loaded at runtime, and you can also customize it.

Yes, this does mean you can change the way the plugin works without rebuilding uWSGI.

A default version of the `pypy_setup.py` file is embedded in the uWSGI binary, and it is automatically loaded on startup.

If you want to change it, just pass another filename via the `--pypy-setup` option.

```
uwsgi --http-socket :9090 --pypy-home /opt/pypy --pypy-lib /opt/libs/libpypy-c.so --pypy-setup /home/
```

This Python module implements uWSGI hooks and the virtual `uwsgi` python module for accessing the uWSGI API from your apps.

If you want to retrieve the contents of the embedded pypy_setup.py file you can read it from the binary symbols with the `print-sym` convenience option.

```
uwsgi --print-sym uwsgi_pypy_setup
```

12.2.7 WSGI support

The plugin implements PEP 333 and PEP 3333. You can load both WSGI modules and `mod_wsgi` style `.wsgi` files.

To load a WSGI module (it must be in your Python path):

```
uwsgi --http-socket :9090 --pypy-home /opt/pypy --pypy-wsgi myapp
```

To load a WSGI file:

```
uwsgi --http-socket :9090 --pypy-home /opt/pypy --pypy-wsgi-file /var/www/myapp/myapp.wsgi
```

12.2.8 RPC support

You can register RPC functions using the `uwsgi.register_rpc()` API function, like you would with the vanilla Python plugin.

```
import uwsgi

def hello():
    return "Hello World"

uwsgi.register_rpc('hello', hello)
```

To call RPC functions, both `uwsgi.rpc()` and `uwsgi.call()` are available.

```
import uwsgi

uwsgi.rpc('192.168.173.100:3031', 'myfunc', 'myarg')
uwsgi.call('myfunc', 'myarg')
uwsgi.call('myfunc@192.168.173.100:3031', 'myarg')
```

Integration (with local RPC) has been tested between PyPy and PyPy, PyPy and JVM, and PyPy and Lua. All of these worked flawlessly... so that means you can call Java functions from PyPy.

12.2.9 IPython trick

Having a runtime shell for making tests is very nice to have. You can use IPython for this.

```
uwsgi --socket :3031 --pypy-home /opt/pypy --pypy-eval "import IPython; IPython.embed()" --honour-sto
```

12.2.10 uWSGI API status

The following API functions, hooks and attributes are supported as of 20130526.

- uwsgi.opt
- uwsgi.post_fork_hook
- uwsgi.add_cron()
- uwsgi.setprocname()
- uwsgi.alarm()
- uwsgi.signal_registered()
- uwsgi.mule_id()
- uwsgi.worker_id()
- uwsgi.masterpid()
- uwsgi.lock()
- uwsgi.unlock()
- uwsgi.add_file_monitor()
- uwsgi.add_timer()
- uwsgi.add_rb_timer()
- uwsgi.cache_get()
- uwsgi.cache_set()
- uwsgi.cache_update()
- uwsgi.cache_del()
- uwsgi.signal()
- uwsgi.call()
- uwsgi.rpc()
- uwsgi.register_rpc()
- uwsgi.register_signal()

12.2.11 Options

- `pypy-lib` - load the specified libpypy-s.so
- `pypy-setup` - load the specified pypy_setup script file
- `pypy-home` - set the pypy home
- `pypy-wsgi` - load a WSGI module
- `pypy-wsgi-file` - load a mod_wsgi compatible .wsgi file
- `pypy-eval` - execute the specified string before `fork()`
- `pypy-eval-post-fork` - execute the specified string after each `fork()`
- `pypy-exec` - execute the specified python script before `fork()`
- `pypy-exec-post-fork` - execute the specified python script after each `fork()`
- `pypy-pp/pypy-python-path/pypy-pythonpath` - add the specified item to the pythonpath
- `pypy-paste` - load a paste.deploy .ini configuration
- `pypy-ini-paste` - load a paste.deploy .ini configuration and use its [uwsgi] section

12.2.12 The alternative approach

The PyPy plugin in uWSGI 1.9.15 has been extended to automatically detect if uWSGI has been called as a shared library by the pypy binary itself (via ctypes for example).

This approach (albeit suboptimal) could be useful to test new pypy releases ('til the PyPy guys start building libpypy-c).

To build a uWSGI shared library with the pypy plugin embedded run:

```
UWSGI_PROFILE=pypy UWSGI_AS_LIB=libuwsgi.so make
```

you will end with a libuwsgi.so shared library you can load with ctypes:

```
#/usr/bin/pypy
import sys
import ctypes
# load the uwsgi library in the global namespace
uwsgi = ctypes.CDLL('./libuwsgi.so',mode=ctypes.RTLD_GLOBAL)

# build command line args
argv = (ctypes.c_char_p * (len(sys.argv)+1))()
pos = 0
for arg in sys.argv:
    argv[pos] = arg
    pos+=1

# inform the uwsgi engine, the passed environ is not safe to overwrite
envs = (ctypes.c_char_p * 1)()

uwsgi.uwsgi_init(len(sys.argv), argv, envs)
```

You can now run the script as a standard uwsgi binary:

```
./fakeuwsgi.py --http-socket :9090 --master --processes 2 --threads 8 --pypy-wsgi myapp
```

as you can see there is no need to specify –pypy-home or –pypy-lib as the pypy environment is already available.

in library mode (under pypy) uWSGI cannot change its process name (under Linux and Solaris, as "environ" is no more valid) and a reload could need a bit of help for finding the right commandline. Generally this trick will be more than enough:

```
./fakeuwsgi.py --http-socket :9090 --master --processes 2 --threads 8 --pypy-wsgi myapp --binary-path
```

12.2.13 Notes

- Mixing libpython with libpypy-c is explicitly forbidden. A check in the pypy plugin prevents you from doing such a hellish thing.
- The PyPy plugin is generally somewhat more "orthodox" from a Python programmer point of view, while the CPython one may be a little blasphemous in many areas. We have been able to make that choice as we do not need backward compatibility with older uWSGI releases.
- The uWSGI API is still incomplete.
- The WSGI loader does not update the uWSGI internal application list, so things like `--need-app` will not work. The server will report "dynamic mode" on startup even if the app has been successfully loaded. This will be fixed soon.

12.3 Running PHP scripts in uWSGI

You can safely run PHP scripts using uWSGI's *CGI* support. The downside of this approach is the latency caused by the spawn of a new PHP interpreter at each request.

To get far superior performance you will want to embed the PHP interpreter in the uWSGI core and use the PHP plugin.

12.3.1 Building

A bunch of distros (such as Fedora, Red Hat and CentOS) include a `php-embedded` package. Install it, along with `php-devel` and you should be able to build the php plugin:

```
python uwsgiconfig --plugin plugins/php
# You can set the path of the php-config script with UWSGICONFIG_PHPPATH.
UWSGICONFIG_PHPPATH=/opt/php53/bin/php-config python uwsgiconfig.py --plugin plugins/php
# or directly specify the directory in which you have installed your php environment
UWSGICONFIG_PHPDIR=/opt/php53 python uwsgiconfig.py --plugin plugins/php
```

If you get linkage problems (such as libraries not found), install those missing packages (`ncurses-devel`, `gmp-devel`, `pcre-devel`...) but be warned that if you add development packages modifying the uWSGI core behaviour (`pcre` is one of these) you _need_ to recompile the uWSGI server too, or strange problems will arise.

For distros that do not supply a libphp package (all Debian-based distros, for instance), you have to rebuild PHP with the `--enable-embed` flag to `./configure`:

```
./configure --prefix=/usr/local --with-mysql --with-mysqli --with-pdo-mysql --with-gd --enable-mbstri
# That's a good starting point
```

Ubuntu 10.04

```
# Add ppa with libphp5-embed package
sudo add-apt-repository ppa:l-mierzwa/lucid-php5
# Update to use package from ppa
sudo apt-get update
# Install needed dependencies
sudo apt-get install php5-dev libphp5-embed libonig-dev libqdbm-dev
# Compile uWSGI PHP plugin
python uwsgiconfig --plugin plugins/php
```

Multiple PHP versions

Sometimes (always, if you are an ISP) you might have multiple versions of PHP installed in the system. In such a case, you will need one uWSGI plugin for each version of PHP:

```
UWSGICONFIG_PHPDIR=/opt/php51 python uwsgiconfig.py --plugin plugins/php default php51
UWSGICONFIG_PHPDIR=/opt/php52 python uwsgiconfig.py --plugin plugins/php default php52
UWSGICONFIG_PHPDIR=/opt/php53 python uwsgiconfig.py --plugin plugins/php default php53
```

'default' is the build profile of your server core. If you build uWSGI without a specific profile, it will be 'default'.

You can then load a specific plugin with `plugins php51`, etc. You cannot load multiple PHP versions in the same uWSGI process.

12.3.2 Running PHP apps with nginx

If you have simple apps (based on file extensions) you can use something like this:

```
location ~ \.php$ {
    root /your_document_root;
    include uwsgi_params;
    uwsgi_modifier1 14;
    uwsgi_pass 127.0.0.1:3030;
}
```

You might want to check for all of URIs containing the string `.php`:

```
location ~ \.php {
    root /your_document_root;
    include uwsgi_params;
    uwsgi_modifier1 14;
    uwsgi_pass 127.0.0.1:3030;
}
```

Now simply run the uWSGI server with a bunch of processes:

```
uwsgi -s :3030 --plugin php -M -p 4
# Or abuse the adaptive process spawning with the --cheaper option
uwsgi -s :3030 --plugin php -M -p 40 --cheaper 4
```

This will allow up to 40 concurrent php requests but will try to spawn (or destroy) workers only when needed, maintaining a minimal pool of 4 processes.

12.3.3 Advanced configuration

By default, the PHP plugin will happily execute whatever script you pass to it. You may want to limit it to only a subset of extensions with the `php-allowed-ext` option.

```
uwsgi --plugin php --master --socket :3030 --processes 4 --php-allowed-ext .php --php-allowed-ext .in
```

12.3.4 Run PHP apps without a frontend server

This is an example configuration with a "public" uWSGI instance running a PHP app and serving static files. It is somewhat complex for an example, but should be a good starting point for trickier configurations.

```
[uwsgi]
; load the required plugins, php is loaded as the default (0) modifier
plugins = http,0:php

; bind the http router to port 80
http = :80
; leave the master running as root (to allows bind on port 80)
master = true
master-as-root = true

; drop privileges
uid = serena
gid = serena

; our working dir
project_dir = /var/www

; chdir to it (just for fun)
chdir = %(project_dir)
; check for static files in it
check-static = %(project_dir)
; ...but skip .php and .inc extensions
static-skip-ext = .php
static-skip-ext = .inc
; search for index.html when a dir is requested
static-index = index.html

; jail our php environment to project_dir
php-docroot = %(project_dir)
; ... and to the .php and .inc extensions
php-allowed-ext = .php
php-allowed-ext = .inc
; and search for index.php and index.inc if required
php-index = index.php
php-index = index.inc
; set php timezone
php-set = date.timezone=Europe/Rome

; disable uWSGI request logging
disable-logging = true
; use a max of 17 processes
processes = 17
; ...but start with only 2 and spawn the others on demand
cheaper = 2
```

A more extreme example that mixes *CGI* with PHP using *internal routing* and a dash of *configuration logic*.

```
[uwsgi]
; load plugins
plugins-dir = /proc/unbit/uwsgi
plugins = cgi,php,router_uwsgi

; set the docroot as a config placeholder
docroot = /accounts/unbit/www/unbit.it

; reload whenever this config file changes
; %p is the full path of the current config file
touch-reload = %p

; set process names to something meaningful
auto-procname = true
procname-prefix-spaced = [unbit.it]

; run with at least 2 processes but increase upto 8 when needed
master = true
processes = 8
cheaper = 2

; check for static files in the docroot
check-static = %(docroot)
; check for cgi scripts in the docroot
cgi = %(docroot)

logto = /proc/unbit/unbit.log
;rotate logs when filesize is higher than 20 megs
log-maxsize = 20971520

; a funny cycle using 1.1 config file logic
for = .pl .py .cgi
  static-skip-ext = %(_)
  static-index = index%(_)
  cgi-allowed-ext = %(_)
endfor =

; map cgi modifier and helpers
; with this trick we do not need to give specific permissions to cgi scripts
cgi-helper = .pl=perl
route = \.pl$ uwsgi:,9,0
cgi-helper = .cgi=perl
route = \.cgi$ uwsgi:,9,0
cgi-helper = .py=python
route = \.py$ uwsgi:,9,0

; map php modifier as the default
route = .* uwsgi:,14,0
static-skip-ext = .php
php-allowed-ext = .php
php-allowed-ext = .inc
php-index = index.php

; show config tree on startup, just to see
; how cool is 1.1 config logic
show-config = true
```

12.3.5 uWSGI API support

Preliminary support for some of the uWSGI API has been added in 1.1. This is the list of supported functions:

- uwsgi_version()
- uwsgi_setprocname($name)
- uwsgi_worker_id()
- uwsgi_masterpid()
- uwsgi_signal($signum)
- uwsgi_rpc($node, $func, ...)
- uwsgi_cache_get($key)
- uwsgi_cache_set($key, $value)
- uwsgi_cache_update($key, $value)
- uwsgi_cache_del($key)

Yes, this means you can call Python functions from PHP using RPC.

```
from uwsgidecorators import *

# define a python function exported via uwsgi rpc api
@rpc('hello')
def hello(arg1, arg2, arg3):
    return "%s-%s-%s" (arg3, arg2, arg1)
```

```
Python says the value is <? echo uwsgi_rpc("", "hello", "foo", "bar", "test"); ?>
```

Setting the first argument of `uwsgi_rpc` to empty, will trigger local rpc.

Or you can share the uWSGI *cache*...

```
uwsgi.cache_set("foo", "bar")
```

```
<? echo uwsgi_cache_get("foo"); ?>
```

12.4 uWSGI Perl support (PSGI)

PSGI is the equivalent of *WSGI* in the Perl world.

- http://plackperl.org/
- https://github.com/plack/psgi-specs/blob/master/PSGI.pod

The PSGI plugin is officially supported and has an officially assigned uwsgi modifier, `5`. So as usual, when you're in the business of dispatching requests to Perl apps, set the `modifier1` value to 5 in your web server configuration.

12.4.1 Compiling the PSGI plugin

You can build a PSGI-only uWSGI server using the supplied `buildconf/psgi.ini` file. Make sure that the `ExtUtils::Embed` module and its prerequisites are installed before building the PSGI plugin.

```
python uwsgiconfig --build psgi
# or compile it as a plugin...
python uwsgiconfig --plugin plugins/psgi
# and if you have not used the default configuration
# to build the uWSGI core, you have to pass
# the configuration name you used while doing that:
python uwsgiconfig --plugin plugins/psgi core
```

or (as always) you can use the network installer:

```
curl http://uwsgi.it/install | bash -s psgi /tmp/uwsgi
```

to have a single-file uwsgi binary with perl support in /tmp/uwsgi

12.4.2 Usage

There is only one option exported by the plugin: `psgi <app>`

You can simply load applications using

```
./uwsgi -s :3031 -M -p 4 --psgi myapp.psgi -m
# or when compiled as a plugin,
./uwsgi --plugins psgi -s :3031 -M -p 4 --psgi myapp.psgi -m
```

12.4.3 Tested PSGI frameworks/applications

The following frameworks/apps have been tested with uWSGI:

- MojoMojo
- Mojolicious
- Mojolicious+perlbrew+uWSGI+nginx install bundle

12.4.4 Multi-app support

You can load multiple almost-isolated apps in the same uWSGI process using the `mount` option or using the `UWSGI_SCRIPT`/`UWSGI_FILE` request variables.

```
[uwsgi]

mount = app1=foo1.pl
mount = app2=foo2.psgi
mount = app3=foo3.pl

server {
  server_name example1.com;
  location / {
    include uwsgi_params;
    uwsgi_pass 127.0.0.1:3031;
    uwsgi_param UWSGI_APPID app1;
    uwsgi_param UWSGI_SCRIPT foo1.pl;
    uwsgi_modifier1 5;
  }
}
```

```
server {
  server_name example2.com;
  location / {
    include uwsgi_params;
    uwsgi_pass 127.0.0.1:3031;
    uwsgi_param UWSGI_APPID app2;
    uwsgi_param UWSGI_SCRIPT foo2.psgi;
    uwsgi_modifier1 5;
  }
}

server {
  server_name example3.com;
  location / {
    include uwsgi_params;
    uwsgi_pass 127.0.0.1:3031;
    uwsgi_param UWSGI_APPID app3;
    uwsgi_param UWSGI_SCRIPT foo3.pl;
    uwsgi_modifier1 5;
  }
}
```

12.4.5 The auto reloader (from uWSGI 1.9.18)

The option –perl-auto-reload <n> allows you to instruct uWSGI to monitor every single module imported by the perl vm.

Whenever one of the module changes, the whole instance will be (gracefully) reloaded.

The monitor works by iterating over %INC after a request is served and the specified number of seconds (from the last run) is elapsed (this number of seconds is the value of the option)

This could look sub-optimal (you wil get the new content starting from from the following request) but it is the more solid (and safe) approach for the way perl works.

If you want to skip specific files from the monitoring, just add them with –perl-auto-reload-ignore

12.4.6 Notes

- Async support should work out-of-the-box.
- Threads are supported on ithreads-enabled perl builds. For each app, a new interpreter will be created for each thread. This shouldn't be too different from a simple multi-process fork()-based subsystem.
- There are currently no known memory leaks.

12.4.7 Real world example, *HTML::Mason*

1. Install the HTML::Mason PSGI handler from CPAN and create a directory for your site.

   ```
   cpan install HTML::Mason::PSGIHandler
   mkdir mason
   ```

2. Create `mason/index.html`:

```
% my $noun = 'World';
% my $ua = $r->headers_in;
% foreach my $hh (keys %{$ua}) {
 <% $hh %><br/>
% }
Hello <% $noun %>!<br/>
How are ya?<br/>
Request <% $r->method %> <% $r->uri %><br/>
```

3. Create the PSGI file (`mason.psgi`):

```
use HTML::Mason::PSGIHandler;

my $h = HTML::Mason::PSGIHandler->new(
        comp_root => "/Users/serena/uwsgi/mason", # required
);

my $handler = sub {
        my $env = shift;
        $h->handle_psgi($env);
};
```

   Pay attention to `comp_root`, it must be an absolute path!

4. Now run uWSGI:

```
./uwsgi -s :3031 -M -p 8 --psgi mason.psgi -m
```

5. Then go to `/index.html` with your browser.

12.5 Ruby support

12.5.1 Ruby API support

Status

The uWSGI API for Ruby is still incomplete (QueueFramework, SharedArea, custom routing and SNMP being the most missing players). The DSL will be extended as soon as the various API calls are ready.

Currently available API functions and constants (available in the UWSGI ruby module) are

- UWSGI.suspend
- UWSGI.masterpid
- UWSGI.async_sleep
- UWSGI.wait_fd_read
- UWSGI.wait_fd_write
- UWSGI.async_connect
- UWSGI.signal
- UWSGI.register_signal
- UWSGI.register_rpc
- UWSGI.signal_registered

- UWSGI.signal_wait
- UWSGI.signal_received
- UWSGI.add_cron
- UWSGI.add_timer
- UWSGI.add_rb_timer
- UWSGI.add_file_monitor
- UWSGI.cache_get
- UWSGI.cache_get!
- UWSGI.cache_exists
- UWSGI.cache_exists?
- UWSGI.cache_del
- UWSGI.cache_set
- UWSGI.cache_set
- UWSGI.cache_set!
- UWSGI.cache_update
- UWSGI.cache_update!
- UWSGI.setprocname
- UWSGI.set_warning_message
- UWSGI.lock
- UWSGI.unlock
- UWSGI.mem
- UWSGI.mule_get_msg
- UWSGI.request_id
- UWSGI.mule_id
- UWSGI.mule_msg
- UWSGI.worker_id
- UWSGI.log
- UWSGI.logsize
- UWSGI.i_am_the_spooler
- UWSGI.send_to_spooler
- UWSGI.spool
- UWSGI::OPT
- UWSGI::VERSION
- UWSGI::HOSTNAME
- UWSGI::NUMPROC
- UWSGI::PIDFILE

- UWSGI::SPOOL_OK
- UWSGI::SPOOL_RETRY
- UWSGI::SPOLL_IGNORE

uWSGI DSL

In parallel to the uWSGI API Python decorators, a DSL for Ruby is available, allowing elegant access to the uWSGI API.

The module is available as `uwsgidsl.rb` in the source distribution. You can put this code in your `config.ru` file, or use the `rbrequire` option to auto-include it.

timer(n, block)

Execute code at regular intervals.

```
timer 30 do |signum|
  puts "30 seconds elapsed"
end
```

rbtimer(n, block)

As timer, but using a red-black tree timer.

```
rbtimer 30 do |signum|
  puts "30 seconds elapsed"
end
```

filemon(path, block)

Execute code at file modifications.

```
filemon '/tmp' do |signum|
  puts "/tmp has been modified"
end
```

cron(hours, mins, dom, mon, dow, block)

Execute a task periodically using the `CronInterface`.

```
cron 20,16,-1,-1,-1 do |signum|
  puts "It's time for tea."
end
```

signal(signum, block)

Register code as a signal handler for the `SignalFramework`.

```
signal 17 do |signum|
  puts "Signal #{signum} was invoked."
end
```

postfork(block)

Execute code after each `fork()`.

```
postfork do
  puts "uWSGI server called fork()"
end
```

rpc(name, block)

Register code as a *uWSGI RPC Stack* function.

```
rpc 'helloworld' do
    return "Hello World"
end

rpc 'advancedhelloworld' do |x,y|
    return "x = #{x}, y = #{y}"
end
```

mule(id?, block)

Execute code as a *Mule* brain.

```
mule 1 do # Run in mule 1
  puts "I am the mule #{UWSGI.mule_id}"
end

mule do # Run in first available mule
  puts "I am the mule #{UWSGI.mule_id}"
end
```

After the function returns, the mule will be brainless. To avoid this, put the code in a loop, or use `muleloop`.

muleloop(id?, block)

Execute code in a mule in looped context.

```
muleloop 3 do
  puts "I am the mule #{UWSGI.mule_id}"
  sleep(2)
end
```

SpoolProc

A subclass of `Proc`, allowing you to define a task to be executed in the *Spooler*.

```
# define the function
my_long_running_task = SpoolProc.new {|args|
  puts "I am a task"
  UWSGI::SPOOL_OK
}

# spool it
my_long_running_task.call({'foo' => 'bar', 'one' => 'two'})
```

MuleFunc

Call a function from any process (such as a worker), but execute in a mule

```
i_am_a_long_running_function = MuleFunc.new do |pippo, pluto|
  puts "i am mule #{UWSGI.mule_id} #{pippo}, #{pluto}"
end

i_am_a_long_running_function.call("serena", "alessandro")
```

The worker calls `i_am_a_long_running_function()` but the function will be execute asynchronously in the first available mule.

If you want to run the function on a specific mule, add an ID parameter. The following would only use mule #5.

```
i_am_a_long_running_function = MuleFunc.new 5 do |pippo,pluto|
  puts "i am mule #{UWSGI.mule_id} #{pippo}, #{pluto}"
end

i_am_a_long_running_function.call("serena", "alessandro")
```

Real world usage

A simple Sinatra app printing messages every 30 seconds:

```
# This is config.ru

require 'rubygems'
require 'sinatra'
require 'uwsgidsl'

timer 30 do |signum|
  puts "30 seconds elapsed"
end

get '/hi' do
  "Hello World!"
end

run Sinatra::Application
```

Or you can put your code in a dedicated file (`mytasks.rb` here)

```
require 'uwsgidsl'

timer 30 do |signum|
  puts "30 seconds elapsed"
end

timer 60 do |signum|
  puts "60 seconds elapsed"
end
```

and then load it with

```
uwsgi --socket :3031 --rack config.ru --rbrequire mytasks.rb --master --processes 4
```

Starting from version 0.9.7-dev a Ruby (Rack/Rails) plugin is officially available. The official modifier number for Ruby apps is 7, so remember to set it in your web server configuration.

Ruby 1.8 and Ruby 1.9 are supported. The plugin can be embedded in the uWSGI core or built as a dynamically loaded plugin.

Some uWSGI standard features still aren't supported by the plugin, such as:

- UDP request management
- *SharedArea – share memory pages between uWSGI components* (support on the way)
- *The uWSGI queue framework*

See the *Ruby API support* page for a list of features currently supported.

12.5.2 Building uWSGI for Ruby support

You can find `rack.ini` in the `buildconf` directory. This configuration will build uWSGI with a Ruby interpreter embedded. To build uWSGI with this configuration, you'll need the Ruby headers/development package.

```
python uwsgiconfig.py --build rack
```

The resulting uWSGI binary can run Ruby apps.

A `rackp.ini` build configuration also exists; this will build uWSGI with Ruby support as a plugin; in this case remember to invoke uWSGI with the `plugins=rack` option.

12.5.3 A note regarding memory consumption

By default the memory management of this plugin is very aggressive (as Ruby can easily devour memory like it was going out of fashion). The Ruby garbage collector is invoked after every request by default. This may hurt your performance if your app creates lots of objects on every request. You can tune the frequency of the collection with the *ruby-gc-freq, rb-gc-freq* option. As usual, there is no one-value-fits-all setting for this, so experiment a bit.

If your app leaks memory without control, consider limiting the number of requests a worker can manage before being restarted with the `max-requests` option. Using `limit-as` can help too.

12.5.4 A note regarding threads and fibers

Adding threading support in Ruby 1.8 is out of discussion. Thread support in this versions is practically useless in a server like uWSGI. Ruby 1.9 has a threading mode very similar to the Python one, its support is available starting from uWSGI 1.9.14 using the "rbthreads" plugin.

Fibers are a new feature of Ruby 1.9. They are an implementation of coroutines/green threads/stop resume/cooperative multithreading, or whatever you'd like to call this class of funny technologies. See `FiberLoop`.

12.5.5 Running Rack applications on uWSGI

This example shows you how to run a Sinatra application on uWSGI.

`config.ru`

```
require 'rubygems'
require 'sinatra'

get '/hi' do
"Hello World!"
end
```

```
run Sinatra::Application
```

Then invoke uWSGI (with `--plugins` if you built Ruby support as a plugin):

```
./uwsgi -s :3031 -M -p 4 -m --post-buffering 4096 --rack config.ru
./uwsgi --plugins rack -s :3031 -M -p 4 -m --post-buffering 4096 --rack config.ru
```

Note: `post-buffering` is required by the Rack specification.

Note: As Sinatra has a built-in logging system, you may wish to disable uWSGI's logging of requests with the `disable-logging` option.

12.5.6 Running Ruby on Rails applications on uWSGI

As writing formal documentation isn't very interesting, here's a couple of examples of Rails apps on uWSGI.

Running Typo

```
sudo gem install typo
typo install /tmp/mytypo
./uwsgi -s :3031 --lazy-apps --master --processes 4 --memory-report --rails /tmp/mytypo --post-buffe
```

–lazy-apps is vital here as typo (like a lot of apps) is not fork-friendly (it does not expect is loaded in the master and then fork() is called). With this option the app is fully loaded one-time per-worker.

Nginx configuration:

```
location / {
  root "/tmp/mytypo/public";
  include "uwsgi_params";
  uwsgi_modifier1 7;
  if (!-f $request_filename) {
    uwsgi_pass 127.0.0.1:3031;
  }
}
```

Running Radiant

```
sudo gem install radiant
radiant /tmp/myradiant
cd /tmp/myradiant
# (edit config/database.yml to fit)
rake production db:bootstrap
./uwsgi -s :3031 -M -p 2 -m --rails /tmp/myradiant --post-buffering 4096 --env RAILS_ENV=production
```

Apache configuration (with static paths mapped directly):

```
DocumentRoot /tmp/myradiant/public

<Directory /tmp/myradiant/public>
  Allow from all
</Directory>
```

```
<Location />
  uWSGISocket 127.0.0.1:3032
  SetHandler uwsgi-handler
  uWSGIForceScriptName /
  uWSGImodifier1 7
</Location>

<Location /images>
  SetHandler default-handler
</Location>

<Location /stylesheets>
  SetHandler default-handler
</Location>

<Location /javascripts>
  SetHandler default-handler
</Location>
```

Rails and SSL

You may wish to use the `HTTPS` / `UWSGI_SCHEME https` uwsgi protocol parameters to inform the app that it is running under HTTPS.

For Nginx:

```
uwsgi_param HTTPS on; # Rails 2.x apps
uwsgi_param UWSGI_SCHEME https; # Rails 3.x apps
```

12.6 Using Lua/WSAPI with uWSGI

Updated for uWSGI 2.0

12.6.1 Building the plugin

The lua plugin is part of the official uWSGI distribution (official modifier 6) and it is availale in the plugins/lua directory.

The plugin support lua 5.1, lua 5.2 and luajit.

By default lua 5.1 is assumed

As always there are various ways to build and install Lua support:

from sources directory:

```
make lua
```

with the installer (the resulting binary will be in /tmp/uwsgi)

```
curl http://uwsgi.it/install | bash -s lua /tmp/uwsgi
```

or you can build it as a plugin

```
python uwsgiconfig.py --plugin plugins/lua
```

or (if you already have a uwsgi binary)

```
uwsgi --build-plugin plugins/lua
```

The build system (check uwsgiplugin.py in plugins/lua directory for more details) uses pkg-config to find headers and libraries.

You can specify the pkg-config module to use with the UWSGICONFIG_LUAPC environment variable.

As an example

```
UWSGICONFIG_LUAPC=lua5.2 make lua
```

will build a uwsgi binary for lua 5.2

as well as

```
UWSGICONFIG_LUAPC=luajit make lua
```

will build a binary with luajit

If you do not want to rely on the pkg-config tool you can manually specify the includes and library directories as well as the lib name with the following environment vars:

12.6.2 Why Lua ?

If you came from other object oriented languages, you may find lua for web development a strange choice.

Well, you have to consider one thing when exploring Lua: it is fast, really fast and consume very few resources.

The uWSGI plugin allows you to write web applications in lua, but another purpose (if not the main one) is using Lua to extend the uWSGI server (and your application) using the signals framework, the rpc subsystem or the simple hooks engine.

If you have slow-area in your code (independently by the language used) consider rewriting them in Lua (before dealing with C) and use uWSGI to safely call them.

12.6.3 Your first WSAPI application

We will use the official WSAPI example, let's call it `pippo.lua`:

```
function hello(wsapi_env)
  local headers = { ["Content-type"] = "text/html" }
  local function hello_text()
    coroutine.yield("<html><body>")
    coroutine.yield("<p>Hello Wsapi!</p>")
    coroutine.yield("<p>PATH_INFO: " .. wsapi_env.PATH_INFO .. "</p>")
    coroutine.yield("<p>SCRIPT_NAME: " .. wsapi_env.SCRIPT_NAME .. "</p>")
    coroutine.yield("</body></html>")
  end
  return 200, headers, coroutine.wrap(hello_text)
end

return hello
```

Now run uWSGI with the `lua` option (remember to add `--plugins lua` as the first command line option if you are using it as a plugin)

```
./uwsgi --http :8080 --http-modifier1 6 --lua pippo.lua
```

This command line starts an http router that forward requests to a single worker in which pippo.lua is loaded.

As you can see the modifier 6 is enforced.

Obviously you can directly attach uWSGI to your frontline webserver (like nginx) and bind it to a uwsgi socket:

```
./uwsgi --socket 127.0.0.1:3031 --lua pippo.lua
```

(remember to set modifier1 to 6 in your webserver of choice)

12.6.4 Concurrency

Basically Lua is available in all of the supported uWSGI concurrency models

you can go multiprocess:

```
./uwsgi --socket 127.0.0.1:3031 --lua pippo.lua --processes 8 --master
```

or multithread:

```
./uwsgi --socket 127.0.0.1:3031 --lua pippo.lua --threads 8 --master
```

or both

```
./uwsgi --socket 127.0.0.1:3031 --lua pippo.lua --processes 4 --threads 8 --master
```

you can run it in coroutine mode (see below) using *uGreen – uWSGI Green Threads* as the suspend engine

```
./uwsgi --socket 127.0.0.1:3031 --lua pippo.lua --async 1000 --ugreen
```

Both threading and async modes will initialize a lua state each (you can see it as a whole independent lua VM)

12.6.5 Abusing coroutines

One of the most exciting feature of Lua are coroutines (cooperative multithreading) support. uWSGI can benefit from this using its async engine. The Lua plugin will initialize a `lua_State` for every async core. We will use a CPU-bound version of our pippo.lua to test it:

```
function hello(wsapi_env)
  local headers = { ["Content-type"] = "text/html" }

  local function hello_text()
    coroutine.yield("<html><body>")
    coroutine.yield("<p>Hello Wsapi!</p>")
    coroutine.yield("<p>PATH_INFO: " .. wsapi_env.PATH_INFO .. "</p>")
    coroutine.yield("<p>SCRIPT_NAME: " .. wsapi_env.SCRIPT_NAME .. "</p>")
    for i=0, 10000, 1 do
        coroutine.yield(i .. "<br/>")
    end
    coroutine.yield("</body></html>")
  end

  return 200, headers, coroutine.wrap(hello_text)
end

return hello
```

and run uWSGI with 8 async cores...

```
./uwsgi --socket :3031 --lua pippo.lua --async 8
```

And just like that, you can manage 8 concurrent requests within a single worker!

Lua coroutines do not work over C stacks (it means you cannot manage them with your C code), but thanks to *uGreen – uWSGI Green Threads* (the uWSGI official coroutine/greenthread engine) you can bypass this limit.

Thanks to uGreen you can use the uWSGI async api in lua apps and gain a very high level of concurrency:

```
uwsgi.async_connect
uwsgi.wait_fd_read
uwsgi.wait_fd_write
uwsgi.is_connected
uwsgi.send
uwsgi.recv
uwsgi.close
uwsgi.ready_fd
```

12.6.6 Threading example

The Lua plugin is "thread-safe" as uWSGI maps a lua_State to each internal pthread. For example you can run the Sputnik wiki engine very easily. Use LuaRocks to install Sputnik and `versium-sqlite3`. A database-backed storage is required as the default filesystem storage does not support being accessed by multiple interpreters concurrently. Create a wsapi compliant file:

```
require('sputnik')
return sputnik.wsapi_app.new{
  VERSIUM_STORAGE_MODULE = "versium.sqlite3",
  VERSIUM_PARAMS = {'/tmp/sputnik.db'},
  SHOW_STACK_TRACE = true,
  TOKEN_SALT = 'xxx',
  BASE_URL       = '/',
}
```

And run your threaded uWSGI server

```
./uwsgi --plugins lua --lua sputnik.ws --threads 20 --socket :3031
```

12.6.7 A note on memory

As we all know, uWSGI is parsimonious with memory. Memory is a precious resource. Do not trust software that does not care for your memory! The Lua garbage collector is automatically called (by default) after each request.

You can tune the frequency of the GC call with the `--lua-gc-freq <n>` option, where n is the number of requests after the GC will be called:

```
[uwsgi]
plugins = lua
socket = 127.0.0.1:3031
processes = 4
master = true
lua = foobar.lua
; run the gc every 10 requests
lua-gc-freq = 10
```

12.6.8 RPC and signals

12.6.9 The Lua shell

12.6.10 Using Lua as 'configurator'

12.6.11 uWSGI api status

12.7 JVM in the uWSGI server (updated to 1.9)

12.7.1 The JWSGI interface

Note: JWSGI is not a standard. Yet. If you like JWSGI, why not send an RFC to the uWSGI mailing list. We have no specific interest in a standard, but who knows...

JWSGI is a port of the WSGI/PSGI/Rack way of thinking for Java.

If, for some obscure reason, you'd feel like developing apps with JVM languages and you don't feel like deploying a huge servlet stack, JWSGI should be up your alley.

It is a very simple protocol: you call a public method that takes a `HashMap` as its sole argument. This HashMap contains CGI style variables and `jwsgi.input` containing a Java InputStream object.

The function has to returns an array of 3 Objects:

- `status` (java.lang.Integer) (example: 200)
- `headers` (HashMap) (example: {"Content-type": "text/html", "Server": "uWSGI", "Foo": ["one","two"]})
- `body` (may be a String, an array of Strings, a File or an InputStream object)

Example

A simple JWSGI app looks like this:

```
import java.util.*;
public class MyApp {

    public static Object[] application(HashMap env) {

        int status = 200;

        HashMap<String,Object> headers = new HashMap<String,Object>();
        headers.put("Content-type", "text/html");
        // a response header can have multiple values
        String[] servers = {"uWSGI", "Unbit"};
        headers.put("Server", servers);

        String body = "<h1>Hello World</h1>" + env.get("REQUEST_URI");

        Object[] response = { status, headers, body };

        return response;
    }
}
```

How to use it ?

You need both the 'jvm' plugin and the 'jwsgi' plugin. A build profile named 'jwsgi', is available in the project to allow a monolithic build with jvm+jwsgi:

```
UWSGI_PROFILE=jwsgi make
```

1. Compile your class with `javac`.

   ```
   javac MyApp.java
   ```

4. Run uWSGI and specify the method to run (in the form class:method)

   ```
   ./uwsgi --socket /tmp/uwsgi.socket --plugins jvm,jwsgi --jwsgi
   MyApp:application --threads 40
   ```

   This will run a JWSGI application on UNIX socket /tmp/uwsgi.socket with 40 threads.

Reading request body

The `jwsgi.input` item is an `uwsgi.RequestBody` object (subclass of java/io/InputStream). You it to access the request body.

```
import java.util.*;
public class MyApp {

    public static Object[] application(HashMap env) {

        int status = 200;

        HashMap<String,Object> headers = new HashMap<String,Object>();
        headers.put("Content-type", "text/plain");

        int body_len = Integer.parseInt((String) env.get("CONTENT_LENGTH"));
        byte[] chunk = new byte[body_len];

        uwsgi.RequestBody input = (uwsgi.RequestBody) env.get("jwsgi.input");

        int len = input.read(chunk);

        System.out.println("read " + len + " bytes");

        String body = new String(chunk, 0, len);

        Object[] response = { status, headers, body };

        return response;
    }
}
```

Pay attention to the use of `read(byte[])` instead of the classical `read()`. The latter inefficiently reads one byte at time, while the former reads a larger chunk at a time.

JWSGI and Groovy

Being low-level, the JWSGI standard can be used as-is in other languages running on the JVM. As an example this is a "Hello World" Groovy example:

```
static def Object[] application(java.util.HashMap env) {
     def headers = ["Content-Type":"text/html", "Server":"uWSGI"]
     return [200, headers, "<h1>Hello World</h1"]
}
```

One serving a static file:

```
static def Object[] application(java.util.HashMap env) {
     def headers = ["Content-Type":"text/plain", "Server":"uWSGI"]
     return [200, headers, new File("/etc/services")]
}
```

The second approach is very efficient as it will abuse uWSGI internal facilities. For example if you have offloading enabled, your worker thread will be suddenly freed. To load Groovy code, remember to compile it:

```
groovyc Foobar.groovy
```

Then run it:

```
./uwsgi --socket /tmp/uwsgi.socket --plugins jvm,jwsgi --jwsgi Foobar:application --threads 40
```

JWSGI and Scala

Like Groovy, you can write JWSGI apps with Scala. You only need the entry point function to use native Java objects:

```
object HelloWorld {
     def application(env:java.util.HashMap[String, Object]): Array[Object] = {
             var headers = new java.util.HashMap[String, Object]()
             headers.put("Content-Type", "text/html")
             headers.put("Server", "uWSGI")
             return Array(200:java.lang.Integer, headers , "Hello World")
     }
}
```

Or in a more Scala-ish way:

```
object HelloWorld {
     def application(env:java.util.HashMap[String, Object]): Array[Object] = {
             val headers = new java.util.HashMap[String, Object]() {
                     put("Content-Type", "text/html")
                     put("Server", Array("uWSGI", "Unbit"))
             }
             return Array(200:java.lang.Integer, headers , "Hello World")
     }
}
```

Once compiled with `scalac <filename>` you run like this:

```
./uwsgi --socket /tmp/uwsgi.socket --plugins jvm,jwsgi --jwsgi HelloWorld:application --threads 40
```

12.7.2 The Clojure/Ring JVM request handler

Thanks to the *JVM in the uWSGI server (updated to 1.9)* plugin available from 1.9, Clojure web apps can be run on uWSGI.

The supported gateway standard is Ring, https://github.com/ring-clojure/ring . Its full specification is available here: https://github.com/ring-clojure/ring/blob/master/SPEC

A uWSGI build profile named "ring" is available for generating a monolithic build with both the JVM and Ring plugins.

From the uWSGI sources:

```
UWSGI_PROFILE=ring make
```

The build system will try to detect your JDK installation based on various presets (for example on CentOS you can `yum install java-1.6.0-openjdk.x86_64-devel` or `java-1.7.0-openjdk-devel.x86_64` or on Debian/Ubuntu `openjdk-6-jdk` and so on...).

OSX/Xcode default paths are searched too.

After a successful build you will have the uwsgi binary and a uwsgi.jar file that you should copy in your CLASSPATH (or just remember to set it in the uwsgi configuration every time).

See also:

For more information on the JVM plugin check *JVM in the uWSGI server (updated to 1.9)*

Our first Ring app

A basic Clojure/Ring app could be the following (save it as myapp.clj):

```
(ns myapp)

(defn handler [req]
     {:status 200
      :headers { "Content-Type" "text/plain" , "Server" "uWSGI" }
      :body (str "<h1>The requested uri is " (get req :uri) "</h1>")
     }
)
```

The code defines a new namespace called 'myapp', in which the 'handler' function is the Ring entry point (the function called at each web request)

We can now build a configuration serving that app on the HTTP router on port 9090 (call it config.ini):

```
[uwsgi]
http = :9090
http-modifier1 = 8
http-modifier2 = 1

jvm-classpath = plugins/jvm/uwsgi.jar
jvm-classpath = ../.lein/self-installs/leiningen-2.0.0-standalone.jar

clojure-load = myapp.clj
ring-app = myapp:handler
```

Run uWSGI:

```
./uwsgi config.ini
```

Now connect to port 9090 and you should see the app response.

As you can note we have manually added uwsgi.jar and the Leiningen standalone jar (it includes the whole Clojure distribution) to our classpath.

Obviously if you do not want to use Leiningen, just add the Clojure jar to your classpath.

The `clojure-load` option loads a Clojure script in the JVM (very similar to what `jvm-class` do with the basic jvm plugin).

The `ring-app` option specify the class/namespace in which to search for the ring function entry point.

In our case the function is in the 'myapp' namespace and it is called 'handler' (you can understand that the syntax is namespace:function)

Pay attention to the modifier configuration. The JVM plugin registers itself as 8, while Ring registers itself as modifier 2 #1, yielding an effective configuration of "modifier1 8, modifier2 1".

Using Leiningen

Leiningen is a great tool for managing Clojure projects. If you use Clojure, you are very probably a Leiningen user.

One of the great advantages of Leiningen is the easy generation of a single JAR distribution. That means you can deploy a whole app with a single file.

Let's create a new "helloworld" Ring application with the `lein` command.

```
lein new helloworld
```

Move it to the just created 'helloworld' directory and edit the project.clj file

```
 (defproject helloworld "0.1.0-SNAPSHOT"
:description "FIXME: write description"
:url "http://example.com/FIXME"
:license {:name "Eclipse Public License"
          :url "http://www.eclipse.org/legal/epl-v10.html"}
:dependencies [[org.Clojure/Clojure "1.4.0"]])
```

We want to add the `ring-core` package to our dependencies (it contains a set of classes/modules to simplify the writing of ring apps) and obviously we need to change the description and URL:

```
 (defproject helloworld "0.1.0-SNAPSHOT"
:description "My second uWSGI ring app"
:url "https://uwsgi-docs.readthedocs.org/en/latest/Ring.html"
:license {:name "Eclipse Public License"
          :url "http://www.eclipse.org/legal/epl-v10.html"}
:dependencies [[org.Clojure/Clojure "1.4.0"] [ring/ring-core "1.2.0-beta1"]])
```

Now save it and run...

```
lein repl
```

This will install all of the jars we need and move us to the Clojure console (just exit from it for now).

Now we want to write our Ring app, just edit the file src/helloworld/core.clj and place the following content in it:

```
(ns helloworld.core
 (:use ring.util.response))

(defn handler [request]
 (-> (response "Hello World")
 (content-type "text/plain")))
```

Then re-edit project.clj to instruct Leiningen on which namespaces to build:

```
 (defproject helloworld "0.1.0-SNAPSHOT"
:description "FIXME: write description"
:url "http://example.com/FIXME"
:license {:name "Eclipse Public License"
          :url "http://www.eclipse.org/legal/epl-v10.html"}
```

```
:aot [helloworld.core]

:dependencies [[org.Clojure/Clojure "1.4.0"] [ring/ring-core "1.2.0-beta1"]])
```

As you can see we have added helloworld.core in the `:aot` keyword.

Now let's compile our code:

```
lein compile
```

And build the full jar (the uberjar):

```
lein uberjar
```

If all goes well you should see a message like this at the end of the procedure:

```
Created /home/unbit/helloworld/target/helloworld-0.1.0-SNAPSHOT-standalone.jar
```

Take a note of the path so we can configure uWSGI to run our application.

```
[uwsgi]
http = :9090
http-modifier1 = 8
http-modifier2 = 1

jvm-classpath = plugins/jvm/uwsgi.jar
jvm-classpath = /home/unbit/helloworld/target/helloworld-0.1.0-SNAPSHOT-standalone.jar

jvm-class = helloworld/core__init

ring-app = helloworld.core:handler
```

This time we do not load Clojure code, but directly a JVM class.

Pay attention: when you specify a JVM class you have to use the '/' form, not the usual dotted form.

The __init suffix is automatically added by the Clojure system when your app is compiled.

The `ring-app` set the entry point to the helloworld.core namespace and the function 'handler'.

We can access that namespace as we have loaded it with `jvm-class`

Concurrency

As all of the JVM plugin request handlers, multi-threading is the best way to achieve concurrency.

Threads in the JVM are really solid, do not be afraid to use them (even if you can spawn multiple processes too)

```
[uwsgi]
http = :9090
http-modifier1 = 8
http-modifier2 = 1

jvm-classpath = plugins/jvm/uwsgi.jar
jvm-classpath = /home/unbit/helloworld/target/helloworld-0.1.0-SNAPSHOT-standalone.jar

jvm-class = helloworld/core__init

ring-app = helloworld.core:handler

master = true
```

```
processes = 4
threads = 8
```

This setup will spawn 4 uWSGI processes (workers) with 8 threads each (for a total of 32 threads).

Accessing the uWSGI api

Clojure can call native Java classes too, so it is able to access the uWSGI API exposed by the JVM plugin.

The following example shows how to call a function (written in python) via Clojure:

```
(ns myapp
 (import uwsgi)
)

(defn handler [req]
  {:status 200
   :headers { "Content-Type" "text/html" , "Server" "uWSGI" }
   :body (str "<h1>The requested uri is " (get req :uri) "</h1>" "<h2>reverse is " (uwsgi/rpc (into-a
  }
)
```

The "reverse" function has been registered from a Python module:

```
from uwsgidecorators import *

@rpc('reverse')
def contrario(arg):
    return arg[::-1]
```

This is the used configuration:

```
[uwsgi]
http = :9090
http-modifier1 = 8
http-modifier2 = 1
jvm-classpath = plugins/jvm/uwsgi.jar
jvm-classpath = /usr/share/java/Clojure-1.4.jar
Clojure-load = myapp.clj
plugin = python
import = pyrpc.py
ring-app = myapp:handler
master = true
```

Another useful feature is accessing the uwsgi cache. Remember that cache keys are string while values are bytes.

The uWSGI Ring implementation supports byte array in addition to string for the response. This is obviously a violation of the standard but avoids you to re-encode bytes every time (but obviously you are free to do it if you like).

Notes and status

- A shortcut option allowing to load compiled code and specifying the ring app would be cool.
- As with the *The JWSGI interface* handler, all of the uWSGI performance features are automatically used (like when sending static files or buffering input)
- The plugin has been developed with the cooperation and ideas of Mingli Yuan. Thanks!

12.7.3 Introduction

As of uWSGI 1.9, you can have a full, thread-safe and versatile JVM embedded in the core. All of the plugins can call JVM functions (written in Java, JRuby, Jython, Clojure, whatever new fancy language the JVM can run) via the *RPC subsystem* or using uWSGI *The uWSGI Signal Framework* The JVM plugin itself can implement request handlers to host JVM-based web applications. Currently *The JWSGI interface* and *The Clojure/Ring JVM request handler* (Clojure) apps are supported. A long-term goal is supporting servlets, but it will require heavy sponsorship and funding (feel free to ask for more information about the project at info@unbit.it).

12.7.4 Building the JVM support

First of all, be sure to have a full JDK distribution installed. The uWSGI build system will try to detect common JDK setups (Debian, Ubuntu, Centos, OSX...), but if it is not able to find a JDK installation it will need some information from the user (see below). To build the JVM plugin simply run:

```
python uwsgiconfig.py --plugin plugins/jvm default
```

Change 'default', if needed, to your alternative build profile. For example if you have a Perl/PSGI monolithic build just run

```
python uwsgiconfig.py --plugin plugins/jvm psgi
```

or for a fully-modular build

```
python uwsgiconfig.py --plugin plugins/jvm core
```

If all goes well the jvm_plugin will be built. If the build system cannot find a JDK installation you will ned to specify the path of the headers directory (the directory containing the jni.h file) and the lib directory (the directory containing libjvm.so). As an example, if jni.h is in /opt/java/includes and libjvm.so is in /opt/java/lib/jvm/i386, run the build system in that way:

```
UWSGICONFIG_JVM_INCPATH=/opt/java/includes UWSGICONFIG_JVM_LIBPATH=/opt/java/lib/jvm/i386 python uwsg
```

After a successful build, you will get the path of the uwsgi.jar file. That jarball containes classes to access the uWSGI API, and you should copy it into your CLASSPATH or at the very least manually load it from uWSGI's configuration.

12.7.5 Exposing functions via the RPC subsystem

In this example we will export a "hello" Java function (returning a string) and we will call it from a Python WSGI application. This is our base configuration (we assume a modular build).

```
[uwsgi]
plugins = python,jvm
http = :9090
wsgi-file = myapp.py
jvm-classpath = /opt/uwsgi/lib/uwsgi.jar
```

The `jvm-classpath` is an option exported by the JVM plugin that allows you to add directories or jarfiles to your classpath. You can specify as many `jvm-classpath` options you need. Here we are manually adding `uwsgi.jar` as we did not copy it into our CLASSPATH. This is our WSGI example script.

```
import uwsgi

def application(environ, start_response):
    start_response('200 OK', [('Content-Type','text/html')])
    yield "<h1>"
```

```
    yield uwsgi.call('hello')
    yield "</h1>"
```

Here we use `uwsgi.call()` instead of `uwsgi.rpc()` as a shortcut (little performance gain in options parsing). We now create our Foobar.java class. Its `static void main()` function will be run by uWSGI on startup.

```
public class Foobar {
   static void main() {

       // create an anonymous function
       uwsgi.RpcFunction rpc_func = new uwsgi.RpcFunction() {
           public String function(String... args) {
               return "Hello World";
           }
       };

       // register it in the uWSGI RPC subsystem
       uwsgi.register_rpc("hello", rpc_func);
   }
}
```

The `uwsgi.RpcFunction` interface allows you to easily write uWSGI-compliant RPC functions. Now compile the Foobar.java file:

```
javac Foobar.java
```

(eventually fix the classpath or pass the uwsgi.jar path with the -cp option) You now have a Foobar.class that can be loaded by uWSGI. Let's complete the configuration...

```
[uwsgi]
plugins = python,jvm
http = :9090
wsgi-file = myapp.py
jvm-classpath = /opt/uwsgi/lib/uwsgi.jar
jvm-main-class = Foobar
```

The last option (`jvm-main-class`) will load a java class and will execute its `main()` method. We can now visit localhost:9090 and we should see the Hello World message.

12.7.6 Registering signal handlers

In the same way as the RPC subsystem you can register signal handlers. You will be able to call Java functions on time events, file modifications, cron... Our Sigbar.java:

```
public class Sigbar {
   static void main() {

       // create an anonymous function
       uwsgi.SignalHandler sh = new uwsgi.SignalHandler() {
           public void function(int signum) {
               System.out.println("Hi, i am the signal " + signum);
           }
       };

       // register it in the uWSGI signal subsystem
       uwsgi.register_signal(17, "", sh);
   }
}
```

`uwsgi.SignalHandler` is the interface for signal handlers.

Whenever signal 17 is rased, the corresponding JVM function will be run. Remember to compile the file, load it in uWSGI and to enable to master process (without it the signal subsystem will not work).

12.7.7 The fork() problem and multithreading

The JVM is not `fork()` friendly. If you load a virtual machine in the master and then you fork() (like generally you do in other languages) the children JVM will be broken (this is mainly because threads required by the JVM are not inherited). For that reason a JVM for each worker, mule and spooler is spawned. Fortunately enough, differently from the vast majority of other platforms, the JVM has truly powerful multithreading support. uWSGI supports it, so if you want to run one of the request handlers (JWSGI, Clojure/Ring) just remember to spawn a number of threads with the `--threads` option.

12.7.8 How does it work?

uWSGI embeds the JVM using the JNI interface. Unfortunately we cannot rely on JVM's automatic garbage collector, so we have to manually unreference all of the allocated objects. This is not a problem from a performance and usage point of view, but makes the development of plugins a bit more difficult compared to other JNI-based products. Fortunately the current API simplifies that task.

12.7.9 Passing options to the JVM

You can pass specific options to the JVM using the `--jvm-opt` option.

For example to limit heap usage to 10 megabytes:

```
[uwsgi]
...
jvm-opt = -Xmx10m
```

12.7.10 Loading classes (without main method)

We have already seen how to load classes and run their `main()` method on startup. Often you will want to load classes only to add them to the JVM (allowing access to external modules needing them) To load a class you can use `--jvm-class`.

```
[uwsgi]
...
jvm-class = Foobar
jvm-class = org/unbit/Unbit
```

Remember class names must use the '/' format instead of dots! This rule applies to `--jvm-main-class` too.

12.7.11 Request handlers

Although the Java(TM) world has its J2EE environment for deploying web applications, you may want to follow a different approach. The uWSGI project implements lot of features that are not part of J2EE (and does not implement lot of features that are a strong part of J2EE), so you may find its approach more suited for your setup (or taste, or skills).

The JVM plugin exports an API to allow hooking web requests. This approach differs a bit from "classic" way uWSGI works. The JVM plugin registers itself as a handler for modifier1==8, but will look at the modifier2 value to know which of its request handlers has to manage it. For example the *The Clojure/Ring JVM request handler* plugin registers itself in the JVM plugin as the modifier2 number '1'. So to pass requests to it you need something like that:

```
[uwsgi]
http = :9090
http-modifier1 = 8
http-modifier2 = 1
```

or with nginx:

```
location / {
    include uwsgi_params;
    uwsgi_modifier1 8;
    uwsgi_modifier2 1;
    uwsgi_pass /tmp/uwsgi.socket;
}
```

Currently there are 2 JVM request handlers available:

- *The JWSGI interface*
- *The Clojure/Ring JVM request handler* (for Clojure)

As already said, the idea of developing a servlet request handler is there, but it will require a sponsorship (aka. money) as it'll be a really big effort.

12.7.12 Notes

- You do not need special jar files to use UNIX sockets – the JVM plugin has access to all of the uWSGI features.
- You may be addicted to the log4j module. There is nothing wrong with it, but do take a look at uWSGI's logging capabilities (less resources needed, less configuration, and more NoEnterprise)
- The uWSGI API access is still incomplete (will be updated after 1.9)
- The JVM does not play well in environments with limited address space. Avoid using `--limit-as` if you load the JVM in your instances.

12.8 The Mono ASP.NET plugin

uWSGI 1.9 added support for the Mono platform, especially for the ASP.NET infrastructure.

The most common way to deploy Mono ASP.NET applications is with the XSP project, a simple web server gateway implementing HTTP and FastCGI protocols.

With the Mono plugin you will be able to host ASP.net applications directly in uWSGI, gaining all of its features in your application for free.

As all of the other uWSGI plugin you can call functions exported from the other languages using the *uWSGI RPC Stack* subsystem.

12.8.1 Building uWSGI + Mono

You can build Mono support as a plugin or in a monolithic build.

A build profile named "mono" is available, making the task pretty simple.

Be sure to have mono installed in your system. You need the Mono headers, the `mcs` compiler and the System.Web assembly. They are available in standard mono distributions.

On recent Debian/Ubuntu systems you can use

```
apt-get install build-essential python mono-xsp4 asp.net-examples
```

`mono-xsp4` is a trick to install all we need in a single shot, as ASP.net examples will be used for testing our setup.

We can build a monolithic uWSGI distribution with Mono embedded:

```
UWSGI_PROFILE=mono make
```

At the end of the procedure (if all goes well) you will get the path to the `uwsgi.dll` assembly.

You may want to install it in your GAC (with gacutil -i <path>) to avoid specifying its path every time. This library allows access to the uWSGI api from Mono applications.

12.8.2 Starting the server

The Mono plugin has an official `modifier1`, 15.

```
[uwsgi]
http = :9090
http-modifier1 = 15
mono-app = /usr/share/asp.net-demos
mono-index = index.asp
```

The previous setup assumes uwsgi.dll has been installed in the GAC, if it is not your case you can force its path with:

```
[uwsgi]
http = :9090
http-modifier1 = 15
mono-app = /usr/share/asp.net-demos
mono-index = index.asp
mono-assembly = /usr/lib/uwsgi/uwsgi.dll
```

`/usr/share/asp.net-demos` is the directory containing Mono's example ASP.net applications.

If starting uWSGI you get an error about not being able to find uwsgi.dll, you can enforce a specific search path with

```
[uwsgi]
http = :9090
http-modifier1 = 15
mono-app = /usr/share/asp.net-demos
mono-index = index.asp
mono-assembly = /usr/lib/uwsgi/uwsgi.dll
env = MONO_PATH=/usr/lib/uwsgi/
```

Or you can simply copy uwsgi.dll into the `/bin` directory of your site directory (`/usr/share/asp.net-demos` in this case).

The `mono-index` option is used to set the file to search when a directory is requested. You can specify it multiple times.

12.8.3 Concurrency and fork() unfriendliness

As the Mono VM is not `fork()` friendly, a new VM is spawned for each worker. This ensures you can run your application in multiprocessing mode.

Mono has really solid multithreading support and it works great with uWSGI's thread support.

```
[uwsgi]
http = :9090
http-modifier1 = 15
mono-app = /usr/share/asp.net-demos
mono-index = index.asp
mono-assembly = /usr/lib/uwsgi/uwsgi.dll
env = MONO_PATH=/usr/lib/uwsgi/

master = true
processes = 4
threads = 20
```

With this setup you will spawn 4 processes each with 20 threads. Try to not rely on a single process. Albeit it is a common setup in the so-called "Enterprise environments", having multiple processes ensures you greater availability (thanks to the master work). This rule (as an example) applies even to the *JVM in the uWSGI server (updated to 1.9)* plugin.

12.8.4 API access

This is a work in progress. Currently only a couple of functions are exported. High precedence will be given to the *uWSGI RPC Stack* and Signal subsystem and to the *The uWSGI caching framework* framework.

12.8.5 Tricks

As always uWSGI tries to optimize (where possible) the "common" operations of your applications. Serving static files is automatically accelerated (or offloaded if offloading is enabled) and all of the path resolutions are cached.

12.9 Running CGI scripts on uWSGI

The CGI plugin provides the ability to run CGI scripts using the uWSGI server.

Web servers/clients/load balancers send requests to the uWSGI server using modifier `9`. uWSGI then uses the variables passed from the client as CGI variables (on occasion fixing them) and calls the corresponding script/executable, re-forwarding its output to the client.

The plugin tries to resemble Apache's behavior, allowing you to run CGI scripts even on webservers that do not support CGI natively, such as Nginx.

12.9.1 Enabling the plugin

The CGI plugin is by default not built in to the core. You need to build a binary with cgi embedded or build the cgi plugin.

To build a single binary with CGI support:

```
curl http://uwsgi.it/install | bash -s cgi /tmp/uwsgi
```

To compile it as a plugin,

```
python uwsgiconfig.py --plugin plugins/cgi
```

or, from sources directory:

```
make PROFILE=cgi
```

12.9.2 Configuring CGI mode

The `cgi <[mountpoint=]path>` option is the main entry point for configuring your CGI environment.

`path` may be a directory or an executable file. In the case of a directory, the CGI plugin will use the URI to find the path of the script. If an executable is passed, it will be run, with `SCRIPT_NAME`, `SCRIPT_FILENAME` and `PATH_INFO` set in its environment.

The `mountpoint` is optional. You can use it to map different URIs to different CGI directories/scripts.

12.9.3 Notes

- Remember to use uWSGI's resource limiting and jailing techniques (namespaces, chroot, capability, unshare....) with your CGI apps to limit the damage they might cause.
- Asynchronous mode is not at all supported with CGI applications. Each CGI application will block the worker running it.
- If not mapped to a helper, each CGI script must have read and execution permissions.

12.9.4 Examples

Example 1: Dumb CGI-enabled directory

```
[uwsgi]
plugins = cgi
socket = uwsgi.sock
cgi = /var/www/cgi-bin
```

Each request will search for the specified file in `/var/www/cgi-bin` and execute it.

A request to `http://example.com/foo.cgi` would run `/var/www/cgi-bin/foo.cgi`.

Example 2: old-style cgi-bin directory

```
[uwsgi]
plugins = cgi
socket = uwsgi.sock
cgi = /cgi-bin=/var/lib/cgi-bin
```

A call to `http://example.com/cgi-bin/foo` will run `/var/lib/cgi-bin/foo`.

Example 3: restricting usage to certain extensions

We want only .cgi and .pl files to be executed:

```
[uwsgi]
plugins = cgi
socket = uwsgi.sock
cgi = /cgi-bin=/var/lib/cgi-bin
cgi-allowed-ext = .cgi
cgi-allowed-ext = .pl
```

Example 4: mapping scripts to interpreters using their extension

We want to run files ending with `.php` in the directory `/var/www` via the `php5-cgi` binary:

```
[uwsgi]
plugins = cgi
socket = uwsgi.sock
cgi = /var/www
cgi-allowed-ext = .php
cgi-helper = .php=php5-cgi
```

If a file is run with an helper, the file to be run will not require the execute permission bit. The helper of course does.

Extension comparison is not case sensitive.

Example 5: running PHP scripts as CGI via Nginx

Configure Nginx to pass .php requests to uWSGI, with `/var/www/foo` as the document root.

```
location ~ .php$ {
  include uwsgi_params;
  uwsgi_param REDIRECT_STATUS 200; # required by php 5.3
  uwsgi_modifier1 9;
  uwsgi_pass 127.0.0.1:3031;
}
```

And configure uWSGI like this:

```
[uwsgi]
plugins = cgi
socket = 127.0.0.1:3031
cgi = /var/www/foo
cgi-allowed-ext = .php
cgi-helper = .php=php5-cgi
```

Example 6: Concurrency

By default each uWSGI worker will be able to run a single CGI script. This mean that using one process, will block your incoming requests until the first request has been ended.

Adding more workers will mitigate the problem, but will consume a lot of memory.

Threads are a better choice. Let's configure each worker process to run 20 worker threads and thus run 20 CGI scripts concurrently.

```
[uwsgi]
plugins = cgi
threads = 20
socket = 127.0.0.1:3031
cgi = /var/www/foo
cgi-allowed-ext = .php
cgi-helper = .php=php5-cgi
```

Starting from uWSGI 2.0.2 you can have even more cheap concurrency thanks to async mode support:

```
[uwsgi]
plugins = cgi
async = 200
ugreen = true
socket = 127.0.0.1:3031
cgi = /var/www/foo
cgi-allowed-ext = .php
cgi-helper = .php=php5-cgi
```

this will spawn 200 coroutines, each able to manage a CGI script (with few K of memory)

Example 7: Mailman web interface behind Nginx

```
location /cgi-bin/mailman {
    include uwsgi_params;
    uwsgi_modifier1 9;
    uwsgi_pass 127.0.0.1:3031;
}
```

```
[uwsgi]
plugins = cgi
threads = 20
socket = 127.0.0.1:3031
cgi = /cgi-bin/mailman=/usr/lib/cgi-bin/mailman
cgi-index = listinfo
```

The `cgi-index` directive specifies which script is run when a path ending with a slash is requested. This way `/cgi-bin/mailman/` will be mapped to the `/cgi-bin/mailman/listinfo` script.

Example 8: Viewvc as CGI in a subdir

Using the Mountpoint option.

```
[uwsgi]
plugins = cgi
threads = 20
socket = 127.0.0.1:3031
cgi = /viewvc=/usr/lib/cgi-bin/viewvc.cgi
```

Example 9: using the uWSGI HTTP router and the `check-static` option

This is pretty much a full-stack solution using only uWSGI running on port 8080.

```
[uwsgi]
plugins = http, cgi

; bind on port 8080 and use the modifier 9
http = :8080
http-modifier1 = 9

; set the document_root as a placeholder
my_document_root = /var/www

; serve static files, skipping .pl and .cgi files
check-static = %(my_document_root)
static-skip-ext = .pl
static-skip-ext = .cgi

; run cgi (ending in .pl or .cgi) in the document_root
cgi = %(my_document_root)
cgi-index = index.pl
cgi-index = index.cgi
cgi-allowed-ext = .pl
cgi-allowed-ext = .cgi
```

Example 10: optimizing CGIs (advanced)

You can avoid the overhead of re-running interpreters at each request, loading the interpreter(s) on startup and calling a function in them instead of `execve()` ing the interpreter itself.

The `contrib/cgi_python.c` file in the source distribution is a tiny example on how to optimize Python CGI scripts.

The Python interpreter is loaded on startup, and after each `fork()`, `uwsgi_cgi_run_python` is called.

To compile the library you can use something like this:

```
gcc -shared -o cgi_python.so -fPIC -I /usr/include/python2.7/ cgi_python.c -lpython2.7
```

And then map `.py` files to the `uwsgi_cgi_run_python` function.

```
[uwsgi]
plugins = cgi

cgi = /var/www
cgi-loadlib = ./cgi_python.so:uwsgi_cgi_load_python
cgi-helper = .py=sym://uwsgi_cgi_run_python
```

}}}

Remember to prefix the symbol in the helper with `sym://` to enable uWSGI to find it as a loaded symbol instead of a disk file.

12.10 The gccgo plugin

uWSGI 1.9.20 officially substituted the old *uWSGI Go support (1.4 only)* plugin with a new one based on gccgo.

The usage of gccgo allows more features and better integration with the uWSGI deployment styles.

A version of the gcc suite >= 4.8 is expected (and strongly suggested)

12.10.1 How it works

when the plugin is enabled a new go runtime is initialized after each fork()

if a `main` Go function is available in the process address space it will be executed in the Go runtime, otherwise the control goes back to the uWSGI loop engine.

12.10.2 Why not plain go ?

Unfortunately the standard go runtime is not (currently) embeddable and does not support compiling code as shared libraries.

Both are requisite for a meaningful uWSGI integration.

Starting from gcc 4.8.2 its libgo has been improved a lot and building shared libraries as well as initializing the Go runtime works like a charm (even if it required a bit of not very elegant hacks)

12.10.3 Building the plugin

A build profile is available allowing you to build a uWSGI+gccgo binary ready to load go shared libraries:

```
make gccgo
```

12.10.4 The first app

You do not need to change the way you write webapps in Go. The net/http package can be used flawlessly:

```
package main

import "uwsgi"
import "net/http"
import "fmt"



func viewHandler(w http.ResponseWriter, r *http.Request) {
     fmt.Fprintf(w, "<h1>Hello World</h1>")
}

func main() {
     http.HandleFunc("/view/", viewHandler)
     uwsgi.Run()
}
```

The only difference is in calling uwsgi.Run() instead of initializing the http go server

To build the code as shared library simply run:

```
gcc -fPIC -shared -o myapp.so myapp.go
```

If you get an error about gcc not able to resolve uwsgi symbols, just add -I<path_to_uwsgi_binary> to the command line (see below):

```
gcc -fPIC -shared -I/usr/bin -o myapp.so myapp.go
```

now let's run it under uWSGI:

```
uwsgi --http-socket :9090 --http-socket-modifier1 11 --go-load ./myapp.so
```

gccgo plugin register itself as modifier1 11, so always remember to set it

12.10.5 uwsgi.gox

By default when building the gccgo profile, a uwsgi.gox file is created. This can be used when building go apps using the uWSGI api, to resolve symbols. Take in account that if you add the directory containing the uwsgi binary (as seen before) to the includes (-I path) path of gcc, the binary itself will be used for resolving symbols

12.10.6 Shared libraries VS monolithic binaries

One of the Go selling point for lot of developers is the "static-all-in-one" binary approach.

Basically a go app does not have dependencies, so half of the common deployments problems automagically disappear.

The uWSGI-friendly way for hosting go apps is having a uWSGI binary loading a specific go app in the form of a library.

If this is not acceptable, you can build a single binary with both uWSGI and the go app:

```
CFLAGS=-DUWSGI_GCCGO_MONOLITHIC UWSGI_ADDITIONAL_SOURCES=myapp.go UWSGI_PROFILE=gccgo make
```

12.10.7 Goroutines

Thanks to the new gcc split stack feature, goroutines are sanely (read: they do not require a full pthread) implemented in gccgo.

A loop engine mapping every uWSGI core to a goroutine is available in the plugin itself.

To start uWSGI in goroutines mode just add `--goroutines <n>` where <n> is the maximum number of concurrent goroutines to spawn.

Like *The Gevent loop engine* uwsgi signal handlers are executed in a dedicated goroutine.

In addition to this all of the blacking calls make use of the netpoll go api (this means you can run internal routing actions, included rpc, in a goroutine)

12.10.8 Options

`--go-load <path>` load the specified go shared library in the process address space

`--gccgo-load <path>` alias for go-load

`--go-args <arg1> <arg2> <argN>` set arguments passed to the virtual go command line

`--gccgo-args <arg1> <arg2> <argN>` alias for go-args

`--goroutines <n>` enable goroutines loop engine with the specified number of async cores

12.10.9 uWSGI API

Unfortunately really few pieces of the uWSGI api have been ported to the gccgo plugin. More features will be added in time for uWSGI 2.0

Currently exposed api functions:

```
uwsgi.CacheGet(key string, cache string) string
```

```
uwsgi.RegisterSignal(signum uint8, receiver string, handler func(uint8)) bool
```

12.10.10 Notes

Do not enable multithreading, it will not work and probably will never work

All of the uWSGI native features (like internal routing) work in goroutines mode, but do not expect languages (like python or perl) to work over them anytime soon.

12.11 The Symcall plugin

The symcall plugin (modifier 18) is a commodity plugin allowing you to write native uWSGI request handlers without the need of developing a full uWSGI plugin.

You tell it which symbol to load on startup and then it will run it at every request.

Note: The "symcall" plugin is builtin by default in standard build profiles

---

12.11.1 Step 1: preparing the environment

The uwsgi binary by itself allows you to develop plugins and library without the need of external development packages or headers.

The first step is getting the `uwsgi.h` C/C++ header:

```
uwsgi --dot-h > uwsgi.h
```

Now, in the current directory, we have uwsgi.h ready to be included.

12.11.2 Step 2: our first request handler:

Our C handler will print the REMOTE_ADDR value with a couple of HTTP headers

(call it mysym.c or whatever you want/need)

```
#include "uwsgi.h"

int mysym_function(struct wsgi_request *wsgi_req) {

     // read request variables
     if (uwsgi_parse_vars(wsgi_req)) {
             return -1;
     }

     // get REMOTE_ADDR
```

```
    uint16_t vlen = 0;
    char *v = uwsgi_get_var(wsgi_req, "REMOTE_ADDR", 11, &vlen);

    // send status
    if (uwsgi_response_prepare_headers(wsgi_req, "200 OK", 6)) return -1;
    // send content_type
    if (uwsgi_response_add_content_type(wsgi_req, "text/plain", 10)) return -1;
    // send a custom header
    if (uwsgi_response_add_header(wsgi_req, "Foo", 3, "Bar", 3)) return -1;

    // send the body
    if (uwsgi_response_write_body_do(wsgi_req, v, vlen)) return -1;

    return UWSGI_OK;
}
```

12.11.3 Step 3: building our code as a shared library

The uwsgi.h file is an ifdef hell.

Fortunately the uwsgi binary exposes all of the required CFLAGS via the –cflags option

We can build our library in one shot:

```
gcc -fPIC -shared -o mysym.so `uwsgi --cflags` mysym.c
```

you now have the mysym.so library ready to be loaded in uWSGI

12.11.4 Final step: map the symcall plugin to the `mysym_function` symbol

```
uwsgi --dlopen ./mysym.so --symcall mysym_function --http-socket :9090 --http-socket-modifier1 18
```

with `--dlopen` we load a shared library in the uWSGI process address space.

the `--symcall` option allows us to specify which symbol to call when modifier1 18 is in place

we bind it to http socket 9090 forcing the modifier1 18

12.11.5 Hooks and symcall unleashed: a TCL handler

We want to write a request handler running the following tcl script (foo.tcl) every time:

```
# call it foo.tcl
proc request_handler { remote_addr path_info query_string } {
    set upper_pathinfo [string toupper $path_info]
    return "Hello $remote_addr $upper_pathinfo $query_string"
}
```

We will define a function for initializing the tcl interpreter and parsing the script. This function will be called on startup soon after privileges drop.

Finally we define the request handler invoking the tcl proc and passign args to it

```
#include <tcl.h>
#include "uwsgi.h"
```

```
// global interpreter
static Tcl_Interp *tcl_interp;

// the init function
void ourtcl_init() {
     // create the tcl interpreter
     tcl_interp = Tcl_CreateInterp() ;
     if (!tcl_interp) {
             uwsgi_log("unable to initialize tcl interpreter\n");
             exit(1);
     }

     // initialize the interpreter
     if (Tcl_Init(tcl_interp) != TCL_OK) {
             uwsgi_log("Tcl_Init error: %s\n", Tcl_GetStringResult(tcl_interp));
             exit(1);
     }

     // parse foo.tcl
     if (Tcl_EvalFile(tcl_interp, "foo.tcl") != TCL_OK) {
             uwsgi_log("Tcl_EvalFile error: %s\n", Tcl_GetStringResult(tcl_interp));
             exit(1);
     }

     uwsgi_log("tcl engine initialized");
}

// the request handler
int ourtcl_handler(struct wsgi_request *wsgi_req) {

     // get request vars
     if (uwsgi_parse_vars(wsgi_req)) return -1;

     Tcl_Obj *objv[4];
     // the proc name
     objv[0] = Tcl_NewStringObj("request_handler", -1);
     // REMOTE_ADDR
     objv[1] = Tcl_NewStringObj(wsgi_req->remote_addr, wsgi_req->remote_addr_len);
     // PATH_INFO
     objv[2] = Tcl_NewStringObj(wsgi_req->path_info, wsgi_req->path_info_len);
     // QUERY_STRING
     objv[3] = Tcl_NewStringObj(wsgi_req->query_string, wsgi_req->query_string_len);

     // call the proc
     if (Tcl_EvalObjv(tcl_interp, 4, objv, TCL_EVAL_GLOBAL) != TCL_OK) {
             // ERROR, report it to the browser
             if (uwsgi_response_prepare_headers(wsgi_req, "500 Internal Server Error", 25)) return -1
             if (uwsgi_response_add_content_type(wsgi_req, "text/plain", 10)) return -1;
             char *body = (char *) Tcl_GetStringResult(tcl_interp);
             if (uwsgi_response_write_body_do(wsgi_req, body, strlen(body))) return -1;
             return UWSGI_OK;
     }

     // all fine
     if (uwsgi_response_prepare_headers(wsgi_req, "200 OK", 6)) return -1;
     if (uwsgi_response_add_content_type(wsgi_req, "text/plain", 10)) return -1;

     // write the result
```

```
    char *body = (char *) Tcl_GetStringResult(tcl_interp);
    if (uwsgi_response_write_body_do(wsgi_req, body, strlen(body))) return -1;
    return UWSGI_OK;
}
```

You can build it with:

```
gcc -fPIC -shared -o ourtcl.so `./uwsgi/uwsgi --cflags` -I/usr/include/tcl ourtcl.c -ltcl
```

(the only differences from the previous example are the -I and -l for adding tcl headers and lib)

and finally run it with:

```
uwsgi --dlopen ./ourtcl.so --hook-as-user call:ourtcl_init --http-socket :9090 --symcall ourtcl_handl
```

here the only new player is `--hook-as-user call:ourtcl_init` invoking the specified function after privileges drop

---

Note: this code is not thread safe, if you want to improve this tcl library to support multithreading, best approach will be having a tcl interpreter for each pthread instead of a global one.

12.11.6 Considerations

Since uWSGI 1.9.21, thanks to the `--build-plugin` option, developing uWSGI plugin became really easy.

The symcall plugin is for tiny libraries/pieces of code, for bigger needs consider developing a full plugin.

The tcl example we have seen before is maybe the right example of "wrong" usage ;)

12.12 The XSLT plugin

Since uWSGI 1.9.1 a new plugin named "xslt" is available, implementing XML Stylesheet Transformation both as request handler and routing instruction.

To successfully apply a transformation you need a 'doc' (an XML document) and a stylesheet (the XSLT file).

Additionally you can apply global params and set a specific content type (by default the generated output is set as text/html).

12.12.1 The request handler

Modifier1 23 has been assigned to the XSLT request handler.

The document path is created appending the `PATH_INFO` to the `DOCUMENT_ROOT`.

The stylesheet path is created following these steps:

- If a specific CGI variable is set (via `--xslt-var`) it will be used as the stylesheet path.
- If a file named like the document plus a specific extension (by default `.xsl` and `.xslt` are searched) exists it will be used as the stylesheet path.
- Finally a series of static XSLT files (specified with `--xslt-stylesheet`) is tried.

Examples:

```
uwsgi --http-socket :9090 --http-socket-modifier1 23 --xslt-ext .bar
```

If /foo.xml is requested (and the file exists) `DOCUMENT_ROOT``+``foo.xml.bar` will be searched as the xslt file.

```
uwsgi --http-socket :9090 --http-socket-modifier1 23 --xslt-stylesheet /var/www/myfile1.xslt --xslt-s
```

If /foo.xml is requested (and the file exists) `/var/www/myfile1.xslt` will be tried. If it does not exist, `/var/www/myfile2.xslt` will be tried instead.

```
uwsgi --http-socket :9090 --http-socket-modifier1 23 --xslt-var UWSGI_XSLT
```

If /foo.xml is requested (and the file exists), the content of the `UWSGI_XSLT` variable (you can set it from your webserver) is used as the stylesheet path.

If a `QUERY_STRING` is available, its items will be passed as global parameters to the stylesheet.

As an example if you request `/foo.xml?foo=bar&test=uwsgi`, "foo" (as "bar" and "test" (as "uwsgi") will be passed as global variables:

```
<xsl:value-of select="$foo"/>
<xsl:value-of select="$test"/>
```

12.12.2 The routing instruction

The plugin registers itself as internal routing instruction named "xslt". It is probably a lot more versatile then the request plugin.

Its syntax is pretty simple:

```
[uwsgi]
plugin = xslt
route = ^/foo xslt:doc=${DOCUMENT_ROOT}/${PATH_INFO}.xml,stylesheet=/var/www/myfunction.xslt,content_
```

This will apply the `/var/www/myfunction.xslt` transformation to `foo.xml` and will return it as `text/html`.

The only required parameters for the routing instruction are `doc` and `stylesheet`.

12.13 SSI (Server Side Includes) plugin

Server Side Includes are an "old-fashioned" way to write dynamic web pages.

It is generally recognized as a templating system instead of a full featured language.

The main purpose of the uWSGI SSI plugin is to have a fast templating system that has access to the uWSGI API.

At the time of writing, March 2013, the plugin is beta quality and implements less than 30% of the SSI standard, the focus being in exposing uWSGI API as SSI commands.

12.13.1 Using it as a request handler

The plugin has an official modifier1, number 19.

```
[uwsgi]
plugin = ssi
http = :9090
```

```
http-modifier1 = 19
http-var = DOCUMENT_ROOT=/var/www
```

The plugin builds the filename as `DOCUMENT_ROOT``+``PATH_INFO`. This file is then parsed as a server side include document.

Both `DOCUMENT_ROOT` and `PATH_INFO` are required, otherwise a 500 error will be returned.

An example configuration for Nginx would be:

```
location ~ \.shtml$ {
    root /var/www;
    include uwsgi_params;
    uwsgi_pass 127.0.0.1:3031;
    uwsgi_modifier1 19;
}
```

with something like this for uWSGI...

```
[uwsgi]
plugin = ssi
socket = 127.0.0.1:3031
```

12.13.2 Using SSI as a routing action

A more versatile approach is using the SSI parser as a routing action.

```
[uwsgi]
plugin = ssi
http-socket = :9090
route = ^/(.*) ssi:/var/www/$1.shtml
```

> **Warning:** As with all of the routing actions, no check on file paths is made to allow a higher level of customization. If you pass untrusted paths to the SSI action, you should sanitize them (you can use routing again, checking for the presence of .. or other dangerous symbols).

And with the above admonition in mind, when used as a routing action, `DOCUMENT_ROOT` or `PATH_INFO` are not required, as the parameter passed contains the full filesystem path.

12.13.3 Supported SSI commands

This is the list of supported commands (and their arguments). If a command is not part of the SSI standard (that is, it's uWSGI specific) it will be reported.

echo

Arguments: `var`

Print the content of the specified request variable.

printenv

Print a list of all request variables.

include

Arguments: `file`

Include the specified file (relative to the current directory).

cache

Note: This is uWSGI specific/non-standard.

Arguments: `key name`

Print the value of the specified cache key in the named cache.

12.13.4 Status

- The plugin is fully thread safe and very fast.
- Very few commands are available, more will be added soon.

12.14 uWSGI V8 support

12.14.1 Building

You will need the `libv8` headers to build the plugin. The official modifier1 value for V8 is '24'.

12.14.2 RPC

```
function part1(request_uri, remote_addr) {
     return '<h1>i am part1 for ' + request_uri + ' ' + remote_addr + "</h1>" ;
}

function part2(request_uri, remote_addr) {
     return '<h2>i am part2 for ' + request_uri + ' ' + remote_addr + "</h2>" ;
}

function part3(request_uri, remote_addr) {
     return '<h3>i am part3 for ' + request_uri + ' ' + remote_addr + "</h3>" ;
}

uwsgi.register_rpc('part1', part1);
uwsgi.register_rpc('part2', part2);
uwsgi.register_rpc('part3', part3);

ciao = function(saluta) {
     uwsgi.log("I have no idea what's going on.");
     return "Ciao Ciao";
}

uwsgi.register_rpc('hello', ciao);
```

12.14.3 Signal handlers

```
function tempo(signum) {
     uwsgi.log("e' passato 1 secondo");
}

uwsgi.register_signal(17, '', tempo);
```

12.14.4 Multitheading and multiprocess

12.14.5 Mules

12.14.6 The uWSGI API

12.14.7 JSGI 3.0

```
exports.app = function (request) {
 uwsgi.log("Hello! I am the app.\n");
 uwsgi.log(request.scheme + ' ' + request.method + ' ' + request.scriptName + ' ' + request.pathInfo
 uwsgi.log(request.serverSoftware);
 return {
     status: 200,
     headers: {"Content-Type": "text/plain", "Server": ["uWSGI", "v8/plugin"]},
     body: ["Hello World!", "I am V8"]
 };
}

uwsgi --plugin v8 --v8-jsgi myapp.js --http-socket :8080 --http-socket-modifier1 24
```

12.14.8 CommonJS

- Require: OK
- Binary/B: NO
- System/1.0: in progress
- IO/A: NO
- Filesystem/A: NO

12.15 The GridFS plugin

Beginning in uWSGI 1.9.5 a "GridFS" plugin is available. It exports both a request handler and an internal routing function. Its official modifier is '25'. The routing instruction is "gridfs" The plugin is written in C++.

12.15.1 Requirements and install

To build the plugin you need the `libmongoclient` headers (and a functioning C++ compiler). On a Debian-like system you can do the following.

```
apt-get install mongodb-dev g++
```

A build profile for gridfs is available:

```
UWSGI_PROFILE=gridfs make
```

Or you can build it as plugin:

```
python uwsgiconfig.py --plugin plugins/gridfs
```

For a fast installation of a monolithic build you can use the network installer:

```
curl http://uwsgi.it/install | bash -s gridfs /tmp/uwsgi
```

This will install a gridfs enabled uwsgi binary.

12.15.2 Standalone quickstart

This is a standalone config that blindly maps the incoming `PATH_INFO` to items in the GridFS db named "test":

```
[uwsgi]
; you can remove the plugin directive if you are using a uWSGI gridfs monolithic build
plugin = gridfs
; bind to http port 9090
http-socket = :9090
; force the modifier to be the 25th
http-socket-modifier1 = 25
; map gridfs requests to the "test" db
gridfs-mount = db=test
```

Assuming you have the myfile.txt file stored in your GridFS as "/myfile.txt", run the following:

```
curl -D /dev/stdout http://localhost:9090/myfile.txt
```

and you should be able to get it.

12.15.3 The initial slash problem

Generally `PATH_INFO` is prefixed with a '/'. This could cause problems in GridFS path resolution if you are not storing the items with absolute path names. To counteract this, you can make the `gridfs` plugin to skip the initial slash:

```
[uwsgi]
; you can remove the plugin directive if you are using a uWSGI gridfs monolithic build
plugin = gridfs
; bind to http port 9090
http-socket = :9090
; force the modifier to be the 25th
http-socket-modifier1 = 25
; map gridfs requests to the "test" db
gridfs-mount = db=test,skip_slash=1
```

Now instead of searching for /myfile.txt it will search for "myfile.txt".

12.15.4 Multiple mountpoints (and servers)

You can mount different GridFS databases under different SCRIPT_NAME (or UWSGI_APPID). If your web server is able to correctly manage the SCRIPT_NAME variable you do not need any additional setup (other than –gridfs-mount). Otherwise don't forget to add the –manage-script-name option

```
[uwsgi]
; you can remove the plugin directive if you are using a uWSGI gridfs monolithic build
plugin = gridfs
; bind to http port 9090
http-socket = :9090
; force the modifier to be the 25th
http-socket-modifier1 = 25
; map gridfs requests to the "test" db
gridfs-mount = db=test,skip_slash=1
; map /foo to db "wolverine" on server 192.168.173.17:4040
gridfs-mount = mountpoint=/foo,server=192.168.173.17:4040,db=wolverine
; map /bar to db "storm" on server 192.168.173.30:4040
gridfs-mount = mountpoint=/bar,server=192.168.173.30:4040,db=storm
; force management of the SCRIPT_NAME variable
manage-script-name = true

curl -D /dev/stdout http://localhost:9090/myfile.txt
curl -D /dev/stdout http://localhost:9090/foo/myfile.txt
curl -D /dev/stdout http://localhost:9090/bar/myfile.txt
```

This way each request will map to a different GridFS server.

12.15.5 Replica sets

If you are using a replica set, you can use it in your uWSGI config with this syntax: <replica>server1,server2,serverN...

```
[uwsgi]
http-socket = :9090
http-socket-modifier1 = 25
gridfs-mount = server=rs0/ubuntu64.local\,raring64.local\,mrspurr-2.local,db=test
```

Pay attention to the backslashes used to escape the server list.

12.15.6 Prefixes

As well as removing the initial slash, you may need to prefix each item name:

```
[uwsgi]
http-socket = :9090
http-socket-modifier1 = 25
gridfs-mount = server=rs0/ubuntu64.local\,raring64.local\,mrspurr-2.local,db=test,prefix=/foobar___
```

A request for /test.txt will be mapped to /foobar___/test.txt

while

```
[uwsgi]
http-socket = :9090
http-socket-modifier1 = 25
gridfs-mount = server=rs0/ubuntu64.local\,raring64.local\,mrspurr-2.local,db=test,prefix=/foobar___,s
```

will map to /foobar___test.txt

12.15.7 MIME types and filenames

By default the MIME type of the file is derived from the filename stored in GridFS. This filename might not map to the effectively requested URI or you may not want to set a `content_type` for your response. Or you may want to allow some other system to set it. If you want to disable MIME type generation just add `no_mime=1` to the mount options.

```
[uwsgi]
http-socket = :9090
http-socket-modifier1 = 25
gridfs-mount = server=ubuntu64.local,db=test,skip_slash=1,no_mime=1
```

If you want your response to set the filename using the original value (the one stored in GridFS) add `orig_filename=1`

```
[uwsgi]
http-socket = :9090
http-socket-modifier1 = 25
gridfs-mount = server=ubuntu64.local,db=test,skip_slash=1,no_mime=1,orig_filename=1
```

12.15.8 Timeouts

You can set the timeout of the low-level MongoDB operations by adding `timeout=N` to the options:

```
[uwsgi]
http-socket = :9090
http-socket-modifier1 = 25
; set a 3 seconds timeout
gridfs-mount = server=ubuntu64.local,db=test,skip_slash=1,timeout=3
```

12.15.9 MD5 and ETag headers

GridFS stores an MD5 hash of each file. You can add this info to your response headers both as ETag (MD5 in hex format) or Content-MD5 (in Base64). Use `etag=1` for adding ETag header and `md5=1` for adding Content-MD5. There's nothing stopping you from adding both headers to the response.

```
[uwsgi]
http-socket = :9090
http-socket-modifier1 = 25
; set a 3 seconds timeout
gridfs-mount = server=ubuntu64.local,db=test,skip_slash=1,timeout=3,etag=1,md5=1
```

12.15.10 Multithreading

The plugin is fully thread-safe, so consider using multiple threads for improving concurrency:

```
[uwsgi]
http-socket = :9090
http-socket-modifier1 = 25
; set a 3 seconds timeout
gridfs-mount = server=ubuntu64.local,db=test,skip_slash=1,timeout=3,etag=1,md5=1
master = true
processes = 2
threads = 8
```

This will spawn 2 processes monitored by the master with 8 threads each for a total of 16 threads.

12.15.11 Combining with Nginx

This is not different from the other plugins:

```
location / {
    include uwsgi_params;
    uwsgi_pass 127.0.0.1:3031;
    uwsgi_modifier1 25;
}
```

Just be sure to set the `uwsgi_modifier1` value to ensure all requests get routed to GridFS.

```
[uwsgi]
socket = 127.0.0.1:3031
gridfs-mount = server=ubuntu64.local,db=test,skip_slash=1,timeout=3,etag=1,md5=1
master = true
processes = 2
threads = 8
```

12.15.12 The 'gridfs' internal routing action

The plugin exports a 'gridfs' action simply returning an item:

```
[uwsgi]
socket = 127.0.0.1:3031
route = ^/foo/(.+).jpg gridfs:server=192.168.173.17,db=test,itemname=$1.jpg
```

The options are the same as the request plugin's, with "itemname" being the only addition. It specifies the name of the object in the GridFS db.

12.15.13 Notes

- If you do not specify a server address, 127.0.0.1:27017 is assumed.
- The use of the plugin in async modes is not officially supported, but may work.
- If you do not get why a request is not serving your GridFS item, consider adding the `--gridfs-debug` option. It will print the requested item in uWSGI logs.

12.16 The GlusterFS plugin

Available from uWSGI 1.9.15

official modifier1: 27

The 'glusterfs' plugin allows you to serve files stored in glusterfs filesystems directly using the glusterfs api available starting from GlusterFS 3.4

This approach (compared to serving via fuse or nfs) has various advantages in terms of performances and ease of deployment.

12.16.1 Step1: glusterfs installation

we build glusterfs from official sources, installing it in /opt/glusterfs on 3 nodes (192.168.173.1, 192.168.173.2, 192.168.173.3).

```
./configure --prefix=/opt/glusterfs
make
make install
```

now start the configuration/control daemon with:

```
/opt/glusterfs/sbin/glusterd
```

from now on we can start configuring our cluster

12.16.2 Step2: the first cluster

run the control client to access the glusterfs shell:

```
/opt/glusterfs/sbin/gluster
```

the first step is "discovering" the other nodes:

```
# do not run on node1 !!!
peer probe 192.168.173.1
# do not run on node2 !!!
peer probe 192.168.173.2
# do not run on node3 !!!
peer probe 192.168.173.3
```

remember, you do not need to run "peer probe" for the same address of the machine on which you are running the glusterfs console. You have to repeat the procedure on each node of the cluser.

Now we can create a replica volume (/exports/brick001 dir has to exist in every node):

```
volume create unbit001 replica 3 192.168.173.1:/exports/brick001 192.168.173.2:/exports/brick001 192
```

and start it:

```
volume start unbit001
```

Now you should be able to mount your glusterfs filesystem and start writing files in it (you can use nfs or fuse)

12.16.3 Step3: uWSGI

a build profile, named 'glusterfs' is already available, so you can simply do:

```
PKG_CONFIG_PATH=/opt/glusterfs/lib/pkgconfig/ UWSGI_PROFILE=glusterfs make
```

The profile currently disable 'matheval' support as the glusterfs libraries use bison/yacc with the same function prefixes (causing nameclash).

You can now start your HTTP serving fastly serving glusterfs files (remember no nfs or fuse are involved):

```
[uwsgi]
; bind on port 9090
http-socket = :9090
; set the default modifier1 to the glusterfs one
http-socket-modifier1 = 27
```

```
; mount our glusterfs filesystem
glusterfs-mount = mountpoint=/,volume=unbit001,server=192.168.173.1:0
; spawn 30 threads
threads = 30
```

12.16.4 High availability

The main GlusterFS selling point is high availability. With the prevopus setup we introduced a SPOF with the control daemon.

The 'server' option allows you to specify multiple control daemons (they are tried until one responds)

```
[uwsgi]
; bind on port 9090
http-socket = :9090
; set the default modifier1 to the glusterfs one
http-socket-modifier1 = 27
; mount our glusterfs filesystem
glusterfs-mount = mountpoint=/,volume=unbit001,server=192.168.173.1:0;192.168.173.2:0;192.168.173.3:0
; spawn 30 threads
threads = 30
```

The '0' port is a glusterfs convention, it means 'the default port' (generally 24007). You can specify whatever port you need/want

12.16.5 Multiple mountpoints

If your webserver (like nginx or the uWSGI http router) is capable of setting protocol vars (like SCRIPT_NAME or UWSGI_APPID) you can mount multiple glusterfs filesystems in the same instance:

```
[uwsgi]
; bind on port 9090
http-socket = :9090
; set the default modifier1 to the glusterfs one
http-socket-modifier1 = 27
; mount our glusterfs filesystem
glusterfs-mount = mountpoint=/,volume=unbit001,server=192.168.173.1:0;192.168.173.2:0;192.168.173.3:0
glusterfs-mount = mountpoint=/foo,volume=unbit002,server=192.168.173.1:0;192.168.173.2:0;192.168.173
glusterfs-mount = mountpoint=/bar,volume=unbit003,server=192.168.173.1:0;192.168.173.2:0;192.168.173
; spawn 30 threads
threads = 30
```

12.16.6 Multiprocess VS multithread

Currently a mix of the both will offers you best performance and availability.

Async support is on work

12.16.7 Internal routing

The *uWSGI internal routing* allows you to rewrite requests to change the requested files. Currently the glusterfs plugin only uses the PATH_INFO, so you can change it via the 'setpathinfo' directive

Caching is supported too. Check the tutorial (linked in the homepage) for some cool idea

12.16.8 Using capabilities (on Linux)

If your cluster requires clients to bind on privileged ports (<1024) and you do not want to change that thing (and obviously you do not want to run uWSGI as root) you may want to give your uWSGI instance the NET_BIND_SERVICE capability. Just ensure you have a capabilities-enabled uWSGI and add

```
... --cap net_bind_service ...
```

to all of the instances you want to connect to glusterfs

12.16.9 Notes:

The plugin automatically enables the mime type engine.

There is no directory index support

12.17 The RADOS plugin

Available from uWSGI 1.9.16

official modifier1: 28

Author: Javier Guerra

The 'rados' plugin allows you to serve objects stored in a Ceph cluster directly using the librados API.

Note that it's not the CephFS filesystem, nor the 'radosgw' S3/Swift-compatible layer; RADOS is the bare object-storage layer.

12.17.1 Step1: Ceph cluster and content

If you want to try a minimal Ceph instalation, you can follow this guide: http://ceph.com/docs/master/start/. note that you only need the OSD and MON daemons, the MDS are needed only for CephFS filesystems.

Once you get it running, you should have a configuration file (by default on /etc/ceph/ceph.con), and should be able to use the *rados* utility.

```
rados lspools
```

by default, you should have at least the 'data', 'metadata' and 'rbd' pools. Now add some content to the 'data' pool. For example, if you have a 'list.html' file and images 'first.jpeg', 'second.jpeg' on a subdirectory 'imgs/':

```
rados -p data put list.html list.html
rados -p data put imgs/first.jpeg imgs/first.jpeg
rados -p data put imgs/second.jpeg imgs/second.jpeg
rados -p data ls -
```

note that RADOS doesn't have a concept of directories, but the object names can contain a slash.

12.17.2 Step2: uWSGI

A build profile, named 'rados' is already available, so you can simply do:

```
python uwsgiconfig.py --build rados
```

You can now start your HTTP to serve RADOS objects:

```
[uwsgi]
; bind on port 9090
http-socket = :9090
; set the default modifier1 to the rados one
http-socket-modifier1 = 28
; mount our rados pool
rados-mount = mountpoint=/rad/,pool=data,config=/etc/ceph/ceph.conf
; spawn 30 threads
threads = 30
```

the 'rados-mount' parameter takes three subparameters:

- mountpoint: required, the URL prefix on which the RADOS objects will appear.
- pool: required, the RADOS pool to serve.
- config: optional, the path to the ceph config file.

in this example, your content will be served at http://localhost:9090/rad/list.html, http://localhost:9090/rad/imgs/first.jpeg and http://localhost:9090/rad/imgs/second.jpeg.

12.17.3 High availability

The RADOS storage system is fully distributed, just starting several uWSGI workers on several machines with the same 'ceph.conf', all will see the same pools. If they all serve on the same mountpoint, you get a failure-resistant RADOS-HTTP gateway.

12.17.4 Multiple mountpoints

You can issue several 'rados-mount' entries, each one will define a new mountpoint. This way you can expose different RADOS pools at different URLs.

12.17.5 Notes:

The plugin automatically enables the mime type engine.

There is no directory index support

Async support is on work

CHAPTER 13

Other plugins

13.1 The Pty plugin

- Available since uWSGI 1.9.15, supported on Linux, OpenBSD, FreeBSD and OSX

This plugin allows you to attach pseudo terminals to your applications.

Currently the pseudoterminal server can be attached (and exposed over network) only on the first worker (this limit will be removed in the future).

The plugin exposes a client mode too (avoiding you to mess with netcat, telnet or screen settings)

13.1.1 Building it

The plugin is not included in the default build profiles, so you have to build it manually:

```
python uwsgiconfig.py --plugin plugins/pty [profile]
```

(remember to specify the build profile if you are not using the default one)

13.1.2 Example 1: Rack application shared debugging

```
UWSGI_PROFILE=ruby2 UWSGI_EMBED_PLUGINS=pty make

./uwsgi --rbshell="require 'pry';binding.pry" --socket /tmp/foo.socket --master --pty-socket :5000

./uwsgi --pty-connect :5000
```

13.1.3 Example 2: IPython control thread

```
import IPython
from uwsgidecorators import *

# only worker 1 has the pty attached
@postfork(1)
@thread
def tshell():
    while True:
        IPython.embed()
```

13.2 SPNEGO authentication

13.3 Configuring uWSGI with LDAP

uWSGI can be configured using LDAP. LDAP is a flexible way to centralize configuration of large clusters of uWSGI servers.

Note: LDAP support must be enabled while `building` uWSGI. The *libldap* library is required.

13.3.1 Importing the uWSGIConfig schema

Running uWSGI with the *–ldap-schema* or *–ldap-schema-ldif* parameter will make it output a standard LDAP schema (or an LDIF file) that you can import into your server.

13.3.2 An example LDIF dump

This is an LDIF dump of an OpenLDAP server with a *uWSGIConfig* entry, running a Trac instance.

```
dn: dc=projects,dc=unbit,dc=it
objectclass: uWSGIConfig
objectclass: domain
dc: projects
uWSGIsocket: /var/run/uwsgi/projects.unbit.it.sock
uWSGIhome: /accounts/unbit/tracvenv
uWSGImodule: trac.web.main:dispatch_request
uWSGImaster: TRUE
uWSGIprocesses: 4
uWSGIenv: TRAC_ENV=/accounts/unbit/trac/uwsgi
```

13.3.3 Usage

You only need to pass a valid LDAP url to the *–ldap* option. Only the first entry returned will be used as configuration.

```
uwsgi –ldap ldap://ldap.unbit.it/dc=projects,dc=unbit,dc=it
```

If you want a filter with sub scope (this will return the first record under the tree *dc=projects,dc=unbit,dc=it* with *ou=Unbit*):

```
uwsgi –ldap ldap://ldap.unbit.it/dc=projects,dc=unbit,dc=it?sub?ou=Unbit
```

CHAPTER 14

Broken/deprecated features

14.1 Integrating uWSGI with Erlang

Warning: Erlang support is broken as of 1.9.20. A new solution is being worked on.

The uWSGI server can act as an Erlang C-Node and exchange messages and RPC with Erlang nodes.

14.1.1 Building

First of all you need the `ei` libraries and headers. They are included in the official erlang tarball. If you are on Debian/Ubuntu, install the `erlang-dev` package. Erlang support can be embedded or built as a plugin. For embedding, add the `erlang` and `pyerl` plugins to your buildconf.

```
embedded_plugins = python, ping, nagios, rpc, fastrouter, http, ugreen, erlang, pyerl
```

or build both as plugins

```
python uwsgiconfig --plugin plugins/erlang
python uwsgiconfig --plugin plugins/pyerl
```

The Erlang plugin will allow uWSGI to became a Erlang C-Node. The `pyerl` plugin will add Erlang functions to the Python plugin.

14.1.2 Activating Erlang support

You only need to set two options to enable Erlang support in your Erlang-enabled uWSGI build. The `erlang` option sets the Erlang node name of your uWSGI server. It may be specified in simple or extended format:

- `nodename@ip`
- `nodename@address`
- `nodename`

The `erlang-cookie` option sets the cookie for inter-node communications. If you do not specify it, the value is taken from the `~/.erlang.cookie` file.

To run uWSGI with Erlang enabled:

```
uwsgi --socket :3031 --erlang testnode@192.168.173.15 --erlang-cookie UUWSGIUWSGIU -p 2
```

14.1.3 A simple RPC hello world example

- Define a new erlang module that exports only a simple function.

```
-module(uwsgitest).
-export([hello/0]).

hello() ->
    'hello world !'.
```

- Launch the `erl` shell specifying the nodename and (eventually) the cookie:

```
erl -name testnode@192.168.173.1
```

- Compile the uwsgitest Erlang module

```
c(uwsgitest).
{ok,uwsgitest}
```

- ... and try to run the `hello` function:

```
uwsgitest:hello().
'hello world !'
```

Great - now that our Erlang module is working, we are ready for RPC! Return to your uWSGI server machine and define a new WSGI module – let's call it `erhello.py`.

```
import uwsgi

def application(env, start_response):
    testnode = uwsgi.erlang_connect("testnode@192.168.173.1")
    start_response('200 OK', [('Content-Type', 'text/plain')])
    yield uwsgi.erlang_rpc(testnode, "uwsgitest", "hello", [])
    uwsgi.erlang_close(testnode)
```

or the fast-style

```
import uwsgi

def application(env, start_response):
    start_response('200 OK', [('Content-Type', 'text/plain')])
    yield uwsgi.erlang_rpc("testnode@192.168.173.1", "uwsgitest", "hello", [])
```

Now relaunch the uWSGI server with this new module:

```
uwsgi --socket :3031 --erlang testnode@192.168.173.15 --erlang-cookie UUWSGIUWSGIU -p 2 -w erhello
```

Point your browser to your uWSGI enabled webserver and you should see the output of your erlang RPC call.

14.1.4 Python-Erlang mappings

The uWSGI server tries to translate Erlang types to Python objects according to the table below.

Python	Erlang	note
str	binary	
unicode	atom	limited by internal atom size
int/long	int	
list	list	
tuple	tuple	
3-tuple	pid	

14.1.5 Sending messages to Erlang nodes

One of the most powerful features of Erlang is the inter-node message passing system. uWSGI can communicate with Erlang nodes as well. Lets define a new Erlang module that simply will echo back whatever we send to it.

```
-module(uwsgiecho).
-export([start/0, loop/0, echo/1]).

echo(Message) ->
        {i_am_echo , Message}.

loop() ->
        receive
                Message1 ->
                        io:format("received a message~n"),
                        { useless, 'testnode@192.168.173.15' } ! echo(Message1)
        end,
        loop().

start() ->
        register(echoer, spawn(uwsgiecho, loop, [])).
```

Remember to register your process with the Erlang `register` function. Using pids to identify processes is problematic. Now you can send messages with `uwsgi.erlang_send_message()`.

```
uwsgi.erlang_send_message(node, "echoer", "Hello echo server !!!" )
```

The second argument is the registered process name. If you do not specify the name, pass a 3-tuple of Python elements to be interpreted as a Pid. If your Erlang server returns messages to your requests you can receive them with `uwsgi.erlang_recv_message()`. Remember that even if Erlang needs a process name/pid to send messages, they will be blissfully ignored by uWSGI.

14.1.6 Receiving erlang messages

Sometimes you want to directly send messages from an Erlang node to the uWSGI server. To receive Erlang messages you have to register "Erlang processes" in your Python code.

```
import uwsgi

def erman(arg):
    print "received an erlang message:", arg

uwsgi.erlang_register_process("myprocess", erman)
```

Now from Erlang you can send messages to the "myprocess" process you registered:

```
{ myprocess, 'testnode@192.168.173.15' } ! "Hello".
```

14.1.7 RPC

You can call uWSGI *uWSGI RPC Stack* functions directly from Erlang.

```
rpc:call('testnode@192.168.173.15', useless, myfunction, []).
```

this will call the "myfunction" uWSGI RPC function on a uWSGI server configured as an Erlang node.

14.1.8 Connection persistence

On high-loaded sites opening and closing connections for every Erlang interaction is overkill. Open a connection on your app initialization with `uwsgi.erlang_connect()` and hold on to the file descriptor.

14.1.9 What about Mnesia?

We suggest you to use Mnesia when you need a high-availability site. Build an Erlang module to expose all the database interaction you need and use `uwsgi.erlang_rpc()` to interact with it.

14.1.10 Can I run EWGI applications on top of uWSGI?

For now, no. The best way to do this would be to develop a plugin and assign a special modifier for EWGI apps.

But before that happens, you can wrap the incoming request into EWGI form in Python code and use `uwsgi.erlang_rpc()` to call your Erlang app.

14.2 Management Flags

Warning: This feature may be currently broken or deprecated.

You can modify the behavior of some aspects of the uWSGI stack remotely, without taking the server offline using the Management Flag system.

Note: A more comprehensive re-setup system may be in the works.

All the flags take an unsigned 32-bit value (so the block size is always 4) that contains the value to set for the flag. If you do not specify this value, only sending the uWSGI header, the server will count it as a read request.

Flag	Action	Description
0	logging	enable/disable logging
1	max_requests	set maximum number of requests per worker
2	socket_timeout	modify the internal socket timeout
3	memory_debug	enable/disable memory debug/report
4	master_interval	set the master process check interval
5	harakiri	set/unset the harakiri timeout
6	cgi_mode	enable/disable cgi mode
7	threads	enable/disable threads (currently unimplemented)
8	reaper	enable/disable process reaper
9	log-zero	enable/disable logging of request with zero response size
10	log-slow	set/unset logging of slow requests
11	log-4xx	enable/disable logging of request with 4xx response status
12	log-5xx	enable/disable logging of request with 5xx response status
13	log-big	set/unset logging of request with big response size
14	log-sendfile	set/unset logging of sendfile requests
15	backlog-status	report the current size of the backlog queue (linux on tcp only)
16	backlog-errors	report the number of errors in the backlog queue (linux on tcp only)

14.2.1 myadmin tool

A simple (and ugly) script, `myadmin`, is included to remotely change management flags:

```
# disable logging on the uWSGI server listening on 192.168.173.17 port 3031
./uwsgi --no-server -w myadmin --pyargv "192.168.173.17:3031 0 0"
# re-enable logging
./uwsgi --no-server -w myadmin --pyargv "192.168.173.17:3031 0 1"
# read a value:
./uwsgi --no-server -w myadmin --pyargv "192.168.173.17:3031 15"
```

14.3 uWSGI Go support (1.4 only)

Warning: Starting from 1.9.20, the Go plugin has been superseded by the *The gccgo plugin* plugin.

Starting from uWSGI 1.4-dev you can host Go web applications in your uWSGI stack. You can download Go from http://golang.org/ . Currently Linux i386/x86_64, FreeBSD i386/x86_64 and OSX are supported. For OSX support, you need Go 1.0.3+ or you will need to apply the patch available at http://code.google.com/p/go/source/detail?r=62b7ebe62958 Goroutines are currently supported only on Linux i386/x86_64.

14.3.1 Building uWSGI with Go support

Go support can be built as an embedded component or plugin. The main difference with the setup of other languages is this time we will build a uwsgi library and not a uwsgi binary. This library will be used by a Go package named uwsgi.go you can link with your apps. Do not be afraid as in the uWSGI distribution there is already a build profile to make a completely (monolithic) distribution with Go support embedded. At the end of the build procedure you will have a libuwsgi.so shared library and a uwsgi.a Go package.

To build uWSGI+go just run (from uWSGI sources directory)

```
UWSGI_PROFILE=go make
```

or if Python is not in your system path, or you need to use a specific python version:

```
/usr/local/bin/python uwsgiconfig.py --build go
```

(or wherever your custom Python is)

At the end of the build procedure you will have a libuwsgi.so file (copy or link it to a library directory like /usr/local/lib or /usr/lib and eventually run ldconfig if needed) and a uwsgi.a file in a subdirectory (based on your arch/os) in plugins/go/pkg.

Important: The last message from the build procedure reports the `GOPATH` you should use when building uWSGI Go apps (copy/remember/annotate that value somewhere).

If you already know how the Go import system works, feel free to copy uwsgi.a in your system-wide GOPATH.

14.3.2 Writing the first Go application

By default the uWSGI Go plugin supports the `http.DefaultServeMux` handler, so if your app is already based on it, running it in uWSGI should be extremely simple.

```
package main

import (
        "uwsgi"
        "net/http"
        "fmt"
)

func oneHandler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "<h1>One</h1>")
}


func twoHandler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "<h2>Two</h2>")
}

func main() {
        http.HandleFunc("/one/", oneHandler)
        http.HandleFunc("/two/", twoHandler)
        uwsgi.Run()
}
```

As you can see, the only differences from a standard `net/http`-based application are the need to `import "uwsgi"` need and calling `uwsgi.Run()` function, which will run the whole uWSGI server. If you want to use your personal request handler instead of `http.DefaultServeMux`, use `uwsgi.Handler(http.Handler)` or `uwsgi.RequestHandler(func(http.ResponseWriter, *http.Request))` to set it.

```
func myHandler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "<h2>Two</h2>")
}

func main() {
        uwsgi.RequestHandler(myHandler)
        uwsgi.Run()
}
```

14.3.3 Building your first app

Assuming you saved your app as helloworld.go, just run the following.

```
GOPATH=/home/foobar/uwsgi/plugins/go go build helloworld.go
```

change GOPATH to the value you got from the build procedure, or to the dir you have installed/copied uwsgi.a If all goes well you will end with a 'helloworld' executable. That executable is a full uWSGI server (yes, really).

```
./helloworld --http :8080 --http-modifier1 11
```

Just point your browser to the port 8080 and check /one/ and /two/ You can start adding processes and a master:

```
./helloworld --http :8080 --http-modifier1 11 --master --processes 8
```

Note: modifier1 11 is officially assigned to Go.

14.3.4 Going in production

In a production environment you will probably put a webserver/proxy in front of your app. Thus your nginx config will look like this:

```
location / {
    include uwsgi_params;
    uwsgi_pass 127.0.0.1:3031;
    uwsgi_modifier1 11;
}
```

while your uWSGI config will be something like this...

```
[uwsgi]
socket = 127.0.0.1:3031
master = true
processes = 4
```

Finally simply run your app:

```
./helloworld config.ini
```

14.3.5 Goroutines (currently Linux/FreeBSD-only)

Goroutines are very probably the most interesting feature of the Go platform. A uWSGI loop engine for goroutines is automatically embedded in the uWSGI library when you build it with the go plugin. To spawn goroutines in each uWSGI process just add the `goroutines = N` option, where N is the number of goroutines to spawn.

```
[uwsgi]
socket = 127.0.0.1:3031
master = true
processes = 4
goroutines = 100
```

With this config you will spawn 100 goroutines for each uWSGI process, yielding a grand total of 400 goroutines (!) As far as uWSGI is concerned, goroutines map to pthreads, but you will be able to spawn coroutine-based tasks from your application as well.

14.3.6 uWSGI api

It is fairly simple to access the uWSGI API from your Go app. To do so invoke the functions exported by the uwsgi package:

```
package main

import (
   "fmt"
   "uwsgi"
)

func hello2(signum int) {
        fmt.Println("I am an rb_timer running on mule", uwsgi.MuleId())
}

func hello(signum int) {
        fmt.Println("Ciao, 3 seconds elapsed")
```

```
}

func postinit() {
        uwsgi.RegisterSignal(17, "", hello)
        uwsgi.AddTimer(17, 3)

        uwsgi.RegisterSignal(30, "mule1", hello2)
        uwsgi.AddRbTimer(30, 5)
}

func foofork() {
   fmt.Println("fork() has been called")
}

func main() {
        uwsgi.PostInit(postinit)
        uwsgi.PostFork(foofork)
        uwsgi.Run()
}
```

The PostInit() function set the 'hook' to be called after the Go initialization is complete. The PostFork() function set the 'hook' to be called after each fork() In postinit hook we register two uwsgi signals, with the second one being run in a mule (the mule1) To run the code just build your new app as above and execute it

```
[uwsgi]
socket = 127.0.0.1:3031
master = true
processes = 2
goroutines = 20
mules = 2
memory-report = true
```

This time we have added memory-report, try it to see how memory-cheap Go apps can be.

14.3.7 Running from the Emperor

If you are running in Emperor mode, you can run uWSGI-Go apps by using the `privileged-binary-patch` option. Your vassal configuration should be something like this.

```
[uwsgi]
socket = 127.0.0.1:3031
master = true
processes = 2
goroutines = 20
mules = 2
memory-report = true
uid = foobar
gid = foobar
privileged-binary-patch = /tmp/bin/helloworld
```

(Obviously change `/tmp/bin/helloworld` to wherever your app lives...)

14.3.8 Notes

- A series of interesting go examples can be found in the `t/go` directory of the uWSGI source distribution.
- Changing process names is currently not possible without modifying the go core

- You cannot use uWSGI native threads with Go (just use –goroutines)
- Only a little part of the uWSGI API has been exposed so far. If you want to hack on it or need more, just edit the uwsgi.go file in the plugins/go/src/uwsgi directory
- Goroutines require the async mode (if you are customizing your uWSGI library remember to always include it)
- It looks like it is possible to load the Python, Lua and PSGI plugins without problems even in goroutines mode (more tests needed)

CHAPTER 15

Release Notes

15.1 Stable releases

15.1.1 uWSGI 2.0.2

Changelog 20140226

Bugfixes

- fixed python3 support on older compilers/libc
- allow starting in spooler-only mode
- fixed cache bitmap support and added test suite (credits: Danila Shtan)
- fixed ftime log var
- added async remote signal management
- fixed end-for and end-if
- fixed loop in internal-routing response chain
- fixed pypy execute_source usage
- logpipe: Don't setsid() twice (credits: INADA Naoki)

New features and improvements

CGI plugin

The plugin has been improved to support streaming.

In addition to this the long-awaited async support is finally ready. Now you can have CGI concurrency without spawning a gazillion of expensive threads/processes

Check: *Running CGI scripts on uWSGI*

PSGI loading improvements

The PSGI loader now tries to use Plack::Util::load_psgi() function instead of simple eval. This addresses various inconsistences in the environment (like the double parsing/compilation/execution of psgi scripts).

If the Plack module is not available, a simple do-based code is used (very similar to load_psgi)

Many thanks to Ævar Arnfjörð Bjarmason of booking.com for having discovered the problem

Availability

uWSGI 2.0.2 can be downloaded from: http://projects.unbit.it/downloads/uwsgi-2.0.2.tar.gz

15.1.2 uWSGI 2.0.1

Changelog [20140209]

Bugfixes and improvements

- due to a wrong prototype declaration, building uWSGI without SSL resulted in a compilation bug. The issue has been fixed.
- a race condition preventing usage of a massive number of threads in the PyPy plugin has been fixed
- check for heartbeat status only if heartbeat subsystem has been enabled
- improved heartbeat code to support various corner cases
- improved psgi.input to support offset in read()
- fixed (and simplified) perl stacktrace usage
- fixed sni secured subscription
- CGI plugin does not require anymore that Status header is the first one (Andjelko Horvat)
- fixed CPython mule_msg_get timeout parsing
- allows embedding of config files via absolute paths
- fixed symcall rpc
- fixed a memory leak in CPython spooler api (xiaost)
- The –no-orphans hardening has been brought back (currently Linux-only)
- improved dotsplit router mode to reduce DOS risk
- sub-Emperor are now loyal by default
- fixed non-shared ruby 1.8.7 support
- fixed harakiri CPython tracebacker
- request vars are now correctly exposed by the stats server
- support log-master for logfile-chown
- improved legion reload
- fixed tuntap netmask
- fixed busyness plugin without metrics subsystem

New features

uWSGI 2.0 is a LTS branch, so do not expect too much new features. 2.0.1 is the first maintainance release, so you still get a bunch of them (mainly features not complete in 2.0)

Perl native Spooler support

Perl finally got full support for the Spooler subsystem. In 2.0 we added server support, in 2.0.1 we completed client support too.

```
use Data::Dumper;

uwsgi::spooler(sub {
     my $env = shift;
     print Dumper($env);
     return uwsgi::SPOOL_OK;
});

uwsgi::spool({'foo' => 'bar', 'arg2' => 'test2'})
```

–alarm-backlog

Raise the specified alarm when the listen queue is full

```
[uwsgi]
alarm = myalarm cmd:mail -s 'ALARM ON QUEUE' admin@example.com
alarm-backlog = myalarm
```

–close-on-exec2

Credits: Kaarle Ritvanen

this flag applies CLOSE_ON_EXEC socket flag on all of the server socket. Use it if you do not want you request-generated processes to inherit the server file descriptor.

Note: –close-on-exec applies the flag on all of the sockets (client and server)

simple notifications subsystem

An annoying problem with subscriptions is that the client does not know if it has been correctly subscribed to the server.

The notification subsystem allows you to add to the subscription packet a datagram address (udp or unix) on which the server will send back messages (like successful subscription)

```
[uwsgi]
; enable the notification socket
notify-socket = /tmp/notify.socket
; pass it in subscriptions
subscription-notify-socket = /tmp/notify.socket
...
```

the notification subsystem is really generic. Expect more subsystem to use it in the future.

pid namespace for daemons (Linux only)

This is a Linux-only, epxerimental feature allowing you to spawn a daemon in a new pid namespace. This feature requires the master running as root.

Check: *Managing external daemons/services*

Resubscriptions

The fastrouter and the http/https/spdy router now support "resubscription".

You can specify a dgram address (udp or unix) on which all of the subscriptions request will be forwarded to (obviously changing the node address to the router one)

The system could be useful to build 'federated' setup:

```
[uwsgi]
fastrouter = 192.168.0.1:3031
fastrouter-subscription-server = 127.0.0.1:5000
fastrouter-resubscribe = 192.168.0.2:5000
```

with this setup the fastrouter on 192.168.0.2 will have all of the records of 192.168.0.1 with the destination set to 192.168.0.1:3031.

filesystem monitor api

The real-time filesystem notification api has been standardized and it is now usable by plugins. The prototype to register a monitor is:

```
struct uwsgi_fsmon *uwsgi_register_fsmon(char *path, void (*func) (struct uwsgi_fsmon *), void *data)
```

it will register a monitor on "path" triggering the function "func" passing "data" as argument.

Remember, this is different from the "touch" api, that is poll-based and can only monitor files. (while fsmon can monitor directories too)

support for yajl 1.0

2.0 added support yajl JSON parser (version 2). 2.0.1 added support for 1.0 too

for-readline

a config-logic iterator that yield file lines:

```
[uwsgi]
for-readline = /etc/myenvs
  env = %(_)
end-for =
```

%i and %j magic vars

%i -> returns the inode of the currently parsed file

%j -> returns hex representation of 32bit djb33x hashing of the currently parsed absolute filename

–inject-before and –inject-after

This two new options should make the config templating system complete for everyone.

They basically prepend and append 'blobs' to a config file.

Yeah, it sound a bit nonsense.

Check the following example:

header.xml:

```
<uwsgi>
    <socket>:3031</socket>
```

footer.xml:

```
<master/>
    </uwsgi>
```

and body.xml:

```
<processes>8</processes>
```

you can build a single config tree with:

```
uwsgi --show-config --inject-before header.xml --inject-after footer.xml --xml body.xml
```

this approach, albeit raw, allows you to use magic-vars in more advanced ways (as you have control on the context of the file using them)

Note: ordering is important, –inject-before and –inject-after must be specified before the relevant config option.

–http-server-name-as-http-host

Some Ruby/Rack middleware make a questionable check on SERVER_NAME/HTTP_HOST matching.

This flag allow the http router to map SERVER_NAME to HTTP_HOST automatically instead of instructing your uWSGI instances to do it.

better Emperor's Ragnarok (shutdown procedure)

The 'Ragnarok' is the Emperor phase executed when you ask him to shutdown.

Before 2.0.1, this procedure simply send KILL to vassals to brutally destroy them.

The new Ragnarok is way more benevolent, asking vassals to gracefully shutdown.

The Emperor tolerance for vassals not shutting down can be tuned with –reload-mercy (default 30 seconds)

PyPy paste support

Two new options for PyPy plugin have been added for paste support:

–pypy-paste <config>

–pypy-ini-paste <ini>

they both maps 1:1 to the CPython variants, but contrary to it they automatically fix logging

Availability

You can download uWSGI 2.0.1 from: http://projects.unbit.it/downloads/uwsgi-2.0.1.tar.gz

15.1.3 uWSGI 2.0

Changelog [20131230]

Important changes

Dynamic options have been definitely removed as well as the broken_plugins directory

Bugfixes and improvements

- improved log rotation
- do not rely on unix signals to print request status during harakiri
- added magic vars for uid and gid
- various Lua fixes
- a tons of coverity-governed bugfixes made by Riccardo Magliocchetti

New features

–attach-daemon2

this is a keyval based option for configuring external daemons.

Updated docs are: *Managing external daemons/services*

Linux setns() support

One of the biggest improvements in uWSGI 1.9-2.0 has been the total support for Linux namespaces.

This last patch adds support for the setns() syscall.

This syscall allows a process to "attach" to a running namespace.

uWSGI instances can exposes their namespaces file descriptors (basically they are the files in /proc/self/ns) via a unix socket.

External instances connects to that unix socket and automatically enters the mapped namespace.

to spawn an instance in "namespace server mode", you use the `--setns-socket <addr>` option

```
uwsgi --setns-socket /var/run/ns.socket --unshare net,ipc,uts ...
```

to attach you simply use `--setns <addr>`

```
uwsgi --setns /var/run/ns.socket ...
```

Updated docs: *Jailing your apps using Linux Namespaces*

"private" hooks

When uWSGI runs your hooks, it verbosely print the whole hook action line. This could be a security problem in some scenario (for example when you run initial phases as root user but allows unprivileged access to logs).

Prefixing your action with a '!' will suppress full logging:

```
[uwsgi]
hook-asap = !exec:my_secret_command
```

Support for yajl library (JSON parser)

Til now uWSGI only supported jansson as the json parser required for managing .js config files.

You can now use the yajl library (available in centos) as alternative JSON parser (will be automatically detected)

Perl spooler support

The perl/PSGI plugin can now be used as a spooler server:

```
uwsgi::spooler(sub {
     my $args = shift;
     print Dumper($args);
     return -2;
});
```

The client part is still missing as we need to fix some internal api problem.

Expect it in 2.0.1 ;)

Gateways can drop privileges

Gateways (like http router, sslrouter, rawrouter, forkptyrouter ...) can now drop privileges independently by the master.

Currently only the http/https/spdy router exposes the new option (`--http-uid/--http-gid`)

Subscriptions-governed SNI contexts

The subscription subsystem now supports 3 additional keys (you can set them with the –subscribe2 option):

`sni_key`

`sni_cert`

`sni_ca`

all of the takes a path to the relevant ssl files.

Check: *SNI - Server Name Identification (virtual hosting for SSL nodes)*

Availability

uWSGI 2.0 has been released on 20131230 and can be downloaded from:

http://projects.unbit.it/downloads/uwsgi-2.0.tar.gz

15.1.4 uWSGI 1.9.21

Latest 1.9 before 2.0 (scheduled at December 30th 2013)

From now on, all of the releases will be -rc's (no new features will be added)

A document describing notes for upgrades from the (extremely obsolete) 1.2 and 1.4 versions is on work.

This release includes a new simplified plugins builder subsystem directly embedded in the uWSGI binary.

A page reporting third plugins is available: *uWSGI third party plugins* (feel free to add yours)

And now....

Changelog [20131211]

Bugfixes

- croak if the psgi streamer fails
- allows building coroae on raspberrypi
- do not wait for write availability until strictly required
- avoid segfault when async mode api is called without async mode
- fixed plain (without suspend engine) async mode
- do not spit errors on non x86 timerfd_create
- support timerfd_create/timerfd_settime on __arm__

Optimizations

writev() for the first chunk

Inernally when the first response body is sent, uWSGI check if response headers have been sent too, and eventually send them with an additional write() call.

This new optimizations allows uWSGI to send both headers and the first body chunk with single writev() syscall.

If the writev() returns with an incomplete write on the second vector, the system will fallback to simple write().

use a single buffer for websockets outgoing packets

Before this patch every single websocket packet required to allocate a memory chunk.

This patch forces the reuse of a single dynamic buffer. For games this should result in a pretty good improvement in responsiveness.

New features

removed zeromq api

The zeromq api (a single function indeed) has been removed. Each plugin rquiring zeromq cam simply call zmq_init() insteadd of uwsgi_zeromq_init().

The mongrel2 support has been moved to a 'mongrel2' plugin.

To pair uWSGI with mongrel2 the same options as before can be used, just remember to load (and build) the mongrel2 plugin

The new sharedarea

The shared area subsystem has been rewritten (it is incompatible with the old api as it requires a new argument as it now supports multiple memory areas).

Check updated docs: *SharedArea – share memory pages between uWSGI components*

report request data in writers and readers

every error when reading and writing to/from the client will report current request's data.

This should simplify debugging a lot.

Modular logchunks management

The uWSGI api has been extended to allow plugins to define their log-request vars.

Check: *Formatting uWSGI requests logs*

tmsecs and tmicros, werr, rerr, ioerr, var.XXX

6 new request logging variables are available:

tmsecs: report the current unix time in milliseconds

tmicros: report the current unix time in microseconds

werr: report the number of write errors for the current request

rerr: report the number of read errors for the current request

ioerr: the sum of werr and rerr

var.XXX: report the context of the request var XXX (like var.PATH_INFO)

mountpoints and mules support for symcall

The symcall plugin has been improved to support mules and mountpoints.

To run a C function in a mule just specify it as `--mule=foobar()` when the mule finds an argument ending with () it will consider it a function symbol.

read2 and wait_milliseconds async hooks

This two non-blocking hooks adds new capabilities to the non-blocking system.

The first one allows to wait on two file descriptors with the same call (currently implemented only in plain async mode)

The second one is used to have a millisecond resolution sleep. (this is currently used only by the sharedarea waiting system)

websockets binary messages

You can now send websocket binary message. Just use `uwsgi.websocket_send_binary()` instead of `uwsgi.websocket_send()`

the 'S' master fifo command

Sending 'S' to the master fifo, enable/disable the sending of subscription packets

as-mule hook

this new custom hooks allows you to execute custom code in every mule:

```
[uwsgi]
hook-as-mule = exec:myscript.sh
...
```

accepting hook and improved chain reloading

The chain reloading subsystem has been improved to take in account when a worker is really ready to accept() requests.

This specific state is announced to the Emperor too.

Check this article for more infos: http://uwsgi-docs.readthedocs.org/en/latest/articles/TheArtOfGracefulReloading.html

–after-request-call

this option allows you to call specific C functions (in chains) after each request. While you should use the framework/interface features for this kind of job, sometimes it is not possibile to execute code after the logging phase. In such a case feel free to abuse this option.

error pages

Three new options allow the definition of custom error pages (html only):

`--error-page-403 <file>` add an error page (html) for managed 403 response

`--error-page-404 <file>` add an error page (html) for managed 404 response

`--error-page-500 <file>` add an error page (html) for managed 500 response

Simplified plugins builder

Building uWSGI plugins is now super easy:

```
uwsgi --build-plugin <directory>
```

this option will create a sane environment based on the current binary (no need to fight with build profiles and #ifdef) and will build the plugin.

No external files (included uwsgi.h) are needed as the uWSGI binary embeds them.

TODO for 2.0

- implement websockets and sharedarea support in Lua
- complete sharedarea api for CPython, Perl, Ruby and PyPy
- implement read2 and wait_milliseconds hook in all of the available loop engines

Availability

uWSGI 1.9.21 has been released on December 11th 2013 and can be downloaded at:

http://projects.unbit.it/downloads/uwsgi-1.9.21.tar.gz

15.1.5 uWSGI 1.9.20

Changelog [20131117]

First round of deprecations and removals for 2.0

- The Go plugin is now considered "broken" and has been moved away from the `plugins` directory. The new blessed way for running Go apps in uWSGI is using *The gccgo plugin* plugin.
- The `--auto-snapshot` option has been removed, advanced management of instances now happens via *The Master FIFO*.
- The matheval support has been removed, while a generic "matheval" plugin (for internal routing) is available (but not compiled in by default). See below for the new way for making "math" in config files.
- The "erlang" and "pyerl" plugins are broken and has been moved out of the `plugins` directory. Erlang support will be completely rewritten after 2.0 release.

Next scheduled deprecations and removals

The ZeroMQ API (a single function indeed) will be removed. Each plugin using ZeroMQ will create its own `zmq` context (no need to share it). This means libzmq will no more be linked in the uWSGI core binary.

Mongrel2 protocol support will be moved to a "mongrel2" plugin instead of being embedded in the core.

Bugfixes

- Fixed master hang when gracefully reloading in lazy mode.
- Fixed `default_app` usage.
- Another round of coverity fixes by Riccardo Magliocchetti.
- Fixed `EAGAIN` management when reading the body.

New features

64bit return values for the RPC subsystem

Before this release every RPC response was limited to a size of 64k (16bit).

Now the RPC protocol automatically detects if more space is needed and can scale up to 64bit.

Another advantage of this approach is that only the required amount of memory per-response is allocated instead of blindly creating a 64k chunk every time.

The new GCCGO plugin

Check official docs: *The gccgo plugin*

The plugin is in early stage of development but it's already quite solid.

Simple math in configuration files

As seen before, we have removed matheval support in favor of a simplified interface:

http://uwsgi-docs.readthedocs.org/en/latest/Configuration.html#placeholders-math-from-uwsgi-1-9-20-dev

For example, now you can automatically set the number of threads to:

```
[uwsgi]
; %k is a magic var translated to the number of cpu cores
threads = %(%k * 3)
...
```

(`%k * 3` is `number_of_cpu_cores * 3`).

New magic vars

`%t` Unix time (in seconds, gathered at instance startup).

`%T` Unix time (in microseconds, gathered at instance startup).

`%k` Number of detected CPU cores.

Perl/PSGI improvements

- *The Chunked input API*.
- `psgix.io` is a `Socket::IO` object mapped to the connection file descriptor (you need to enable it with `--psgi-enable-psgix-io`).
- `uwsgi::rpc` and `uwsgi::connection_fd` from the API.
- `--plshell` will invoke an interactive shell (based on `Devel::REPL`).

New native protocols: `--https-socket` and `--ssl-socket`

When built with SSL support, uWSGI exposes two new native socket protocols: HTTPS and uwsgi over SSL.

Both options take the following value: `<addr>,<cert>,<key>[,ciphers,ca]`.

```
[uwsgi]
https-socket = :8443,foobar.crt,foobar.key
...
```

Currently none of the mainstream webservers support uwsgi over SSL, a patch for nginx will be sent for approval in the next few hours.

PROXY (version1) protocol support

Recently Amazon ELB added support for HAProxy PROXY (version 1) protocol support. This simple protocol allows the frontend to pass the real IP of the client to the backend.

Adding `--enable-proxy-protocol` will force the `--http-socket` to check for a PROXY protocol request for setting the `REMOTE_ADDR` and `REMOTE_PORT` fields.

New metrics collectors

`avg` Compute the math average of children: `--metric name=foobar,collector=avg,children=metric1;metric2`.

`accumulator` Always add the value of the specified children to the final value.

`multiplier` Multiply the sum of the specified children for the value specified in `arg1n`.

Check *The Metrics subsystem*.

Availability

uWSGI 1.9.20 has been released on 20131117 and can be downloaded from http://projects.unbit.it/downloads/uwsgi-1.9.20.tar.gz.

15.1.6 uWSGI 1.9.19

Changelog [20131109]

This release starts the 'hardening' cycle for uWSGI 2.0 (scheduled for the end of december 2013).

The metrics subsystem was the last piece missing and this version (after 1 year of analysis) finally includes it.

During the following 2 months we will start deprecating features or plugins that got no-interest, are known to be broken or are simply superseed by more modern/advanced ones.

Currently the following plugin and features are scheduled for removal:

- The Go plugin, superseeded by the gccgo one. (eventually the Go plugin will be brought back if something changes in the fork() support)
- Auto-snapshotting, was never documented, it has tons of corner case bugs and it is huber-complex. The features added by the `MasterFifo` allows for better implementations of snapshotting.

Waiting for decision:

- the erlang plugin is extremely old, was badly engineered and should be completely rewritten. If you are a user of it, please contact the staff. Very probably we will not be able to maintain it without sponsorship.
- the matheval support could be removed soon (unless we find some specific use that could require it), substituted by some form of simple math directly implemented in the option parser
- the admin plugin should be substituted with something more advanced. An api for defining dynamic options is on-work

Bugfixes

- completely skip cgroups initialization when non-root
- tons of post-static_analysis fixes by Riccardo Magliocchetti
- fixed the greenlet plugin reference counting
- avoid kevent storm for stats pusher thread
- fixed rbtimers math
- both 'cache' and 'file' routers got a 'no_content_length' key option to avoid settign the Content-Length header
- the PyPy plugin automatically enables threads/GIL
- manage dot_segments in HTTP parser
- improved srand() usage

New features

The Metrics subsystem

This was the last piece missing before uWSGI 2.0. The Metrics subsystem allows you to store "numbers" related to monitoring, graphing and quality checks and exports them in various ways.

Official docs: *The Metrics subsystem*

The Tornado loop engine

While working on nodejs integration we realized that contrary to what we used to believe, Tornado (an asynchronous, callback based module for python) is usable in uWSGI.

Note: The plugin is not built-in by default

Official docs: *The Tornado loop engine*

The 'puwsgi' protocol

A "persistent" (keep-alive) version of the 'uwsgi' parser has been added named 'puwsgi' (persistent uwsgi).

This protocol works only for request without a body and requires support from the frontend. Its use is currently for custom clients/apps, there is no webserver handler supporting it.

The `--puwsgi-socket <addr>` will bind a puwsgi socket to the specified address

–vassal-set

You can tell the Emperor to pass specific options to every vassal using the –set facility:

```
[uwsgi]
emperor = /etc/uwsgi/vassals
vassal-set = processes=8
vassal-set = enable-metrics=1
```

this will add `--set processes=8` and `--set enable-metrics=1` to each vassal

The 'template' transformation

This is a transformation allowing you to apply all of the internal routing patterns to your responses.

Take the following file (foo.html)

```
<html>
  <head>
    <title>Running on ${SERVER_NAME}</title>
  </head>
  <body>
    Your ip address is: ${REMOTE_ADDR}<br/>
    Served requests: ${metric[worker.0.requests]}<br/>
    Pid: ${uwsgi[pid]}<br/>
    A random UUID: ${uwsgi[uuid]}
  </body>
</html>
```

we will apply the 'template' transformation to it:

```
[uwsgi]
http-socket = :9090
; enable the metrics subsystem
enable-metrics = true
; inject the route transformation
route-run = template:
; return a file (transformation will be applied to it)
route-run = file:filename=foo.html,no_content_length=1
```

everything available in the internal routing subsystem can be used into the template transformation.

Performance are stellar, so instead of old Server Side Includes, you may want to try it.

Not enough ? combine it with caching:

```
[uwsgi]
http-socket = :9090
; enable the metrics subsystem
enable-metrics = true
; load foo.html in the cache
cache2 = name=mycache,items=10
load-file-in-cache = foo.html
; inject the route transformation
route-run = template:
; return the cache item (transformation will be applied to it)
route-run = cache:key=foo.html,no_content_length=1
```

Again ?

what about chunked encoding ?

```
[uwsgi]
http-socket = :9090
; enable the metrics subsystem
enable-metrics = true
; load foo.html in the cache
cache2 = name=mycache,items=10
load-file-in-cache = foo.html
; inject the route transformation
route-run = template:
; inject chunked encoding
route-run = chunked:
; return the cache item (transformation will be applied to it)
route-run = cache:key=foo.html,no_content_length=1
```

or gzip ?

```
[uwsgi]
http-socket = :9090
; enable the metrics subsystem
enable-metrics = true
; load foo.html in the cache
cache2 = name=mycache,items=10
load-file-in-cache = foo.html
; inject the route transformation
route-run = template:
; inject gzip
route-run = gzip:
; return the cache item (transformation will be applied to it)
route-run = cache:key=foo.html,no_content_length=1
```

Availability

uWSGI 1.9.19 has been released on 20131109, you can download it from:

http://projects.unbit.it/downloads/uwsgi-1.9.19.tar.gz

15.1.7 uWSGI 1.9.18

Changelog [20131011]

License change

This version of uWSGI is the first of the 1.9 tree using GPL2 + linking exception instead of plain GPL2.

This new license should avoid any problems when using uWSGI as a shared library (or when linking it with non-GPL2 compatible libraries)

Remember: if you need to make closed-source modifications to uWSGI you can buy a commercial license.

Bugfixes

- fixed uwsgi native protocol support on big endian machines
- fixed jvm build system for arm (Jorge Gallegos)

- fixed a memleak spotted by cppcheck in zlib management
- chdir() at every emperor glob iteration
- correctly honour –force-cwd
- fixed ia64/Linux compilation (Jonas Smedegaard/Riccardo Magliocchetti)
- fixed ruby rvm paths parsing order
- added waitpid() after daemon's SIGTERM (Łukasz Mierzwa)
- fixed pid numbering after –idle (Łukasz Mierzwa)
- fixed/improved cheaper memory limits (Łukasz Mierzwa)
- correctly close inherited sockets in gateways
- fix checks for MAP_FAILED in mmap() (instead of NULL)
- fixed FastCGI non-blocking body read() (patch by Arkaitz Jimenez)
- fixed attach.py script
- avoid crashing on non-conformant PSGI response headers
- run the python autoreloader even in non-apps mode when non-lazy

New Features

Minimal build profiles

Albeit the memory usage of the uWSGI core is generally between 1.8 and 2.5 megs, there are use cases in which you want an even minimal core and set of embedded plugins.

Examples are users not making use of uWSGI specific features, or cases in which the libraries used by uWSGI nameclash with others (like openssl or zeromq).

A bunch of 'minimal' build profiles have been added:

- pyonly (build a minimal CPython WSGI server)
- pypyonly (build a minimal PyPy WSGI server)
- plonly (build a minimal PSGI server)
- rbonly (build a minimal Rack server)

the only supported configuration format is .ini and internal routing and legion subsystem are not builtin.

For example if you want to install a minimal uWSGI binary via pip:

```
UWSGI_PROFILE=pyonly pip install uwsgi
```

IMPORTANT: minimal build profiles do not improve performance, for the way uWSGI is designed, unused features do not waste CPU. Minimal build profiles impact on final binary size only

Auto-fix modifier1

Setting the modifier1 for non-python plugin is pretty annoying (read: you always forget about it).

Now if the modifier1 of the request is zero, but the python plugin is not loaded (or there are no python apps loaded) the first configured app will be set instead (unless you disable with feature with –no-default-app).

This means you can now run:

```
uwsgi --http-socket :9090 --psgi myapp.pl
```

instead of

```
uwsgi --http-socket :9090 --http-socket-modifier1 5 --psgi myapp.pl
```

obviously try to always set the modifier1, this is only a handy hack

Perl auto reloader

The –perl-auto-reload option allows the psgi plugin to check for changed modules after every request. It takes the frequency (in seconds) of the scan.

The scan happens after a request has been served. It is suboptimal, but it is the safest choice too.

The "raw" mode (preview technology, only for CPython)

While working on a new server-side project in Unbit we had the need to expose our web application using a very specific protocol (none of the ones supported by uWSGI).

Our first way was adding the new protocol as a plugin, but soon we realize that is was too-specific. So we decided to introduce the RAW mode.

Raw mode allows you to directly parse the request in your application callable. Instead of getting a list of CGI vars/headers in your callable you only get the file descriptor soon after accept().

You can then read()/write() to that file descriptor in full freedom.

```
import os
def application(fd):
   os.write(fd, "Hello World")
```

```
uwsgi --raw-socket :7070 --python-raw yourapp.py
```

Raw mode disables request logging. We currently support it only for CPython, if we get reports (or interest) about it for the other languages we will add support for sure.

IMPORTANT: raw mode is not a standard, so do not expect any middleware or common usage patterns will apply. Use it as a low-level socket wrapper.

Optional NON-standard support for CPython buffer protocol for WSGI responses

Authors: yihuang with help of INADA Naoki (methane)

The WSGI (PEP333/3333) is pretty clear about the type of valid objects for responses: str for python2, bytes for python3

uWSGI (heavily using mod_wsgi as a reference) always enforce such behaviour, so "exotic" patterns like returning bytearray where not supported. Such uses are somewhat involuntary supported on pure-python application servers, just because they simply call write() over them or because they cast them to string before returning (very inefficient)

The patch proposed by yihuang suggests the use of the low-level buffer protocol exposed by the CPython C api. Strings (in python2) and bytes (in python3) support the buffer protocol, so its use is transparent and backward compatibility is granted too. (for the CPython C api experts: yes we support both old and new buffer protocol)

This is a NON-standard behaviour you have to voluntary enable with –wsgi-accept-buffer.

Use with care as it could mask errors and/or wrong behaviours.

Note: if you tried 1.9.18-dev you may note this option was enabled by default. It was an error. Thanks to Graham Dumpleton (mod_wsgi author) for pointing it out.

Emperor and config improvements

Credits: Matthijs Kooijman

The config system has been improved to be even more consistent in respect to strict mode (remainder: with –strict you basically check your config files for unknown options avoiding headaches caused by typos).

New magic vars have been added exposing the name of the original config file (this simplify templating when in Emperor mode), check them at https://github.com/unbit/uwsgi-docs/blob/master/Configuration.rst#magic-variables

The Emperor got support for Linux capabilities using the –emperor-cap option. The option takes the list of capability you want to maintain for your vassals when they start as root:

```
[uwsgi]
emperor = /etc/uwsgi/vassals
emperor-cap = setuid,net_bind_service
```

with this setup your vassal will be only able to drop privileges and bind to ports < 1024

Its best friend is the CLONE_NEWUSER flag of linux namespaces that is now fully supported on uWSGI:

```
[uwsgi]
emperor = /etc/uwsgi/vassals
emperor-use-clone = user
emperor-cap = setuid,net_bind_service
```

this will create a new root user for the vassal with fewer privileges (CLONE_NEWUSER is pretty hard to understand, but the best thing to catch it is seeing it as a new root user with dedicated capabilities)

Build system improvements

The build system has been improved to link custom sources on the fly. This works great for low-level hooks:

```
// embed_me.c
#include <stdio.h>

void hello_i_am_foobar() {
        printf("I Am foobar");
}
```

Now we can link this file to the main uWSGI binary in one shot:

```
UWSGI_ADDITIONAL_SOURCES=embed_me.c make
```

and you will automatically get access for your hooks:

```
uwsgi --http-socket :9090 --call-asap hello_i_am_foobar
```

Finally, Riccardo Magliocchetti rewrote the build script to use optparse instead of raw/old-fashioned sys.argv parsing

Pluginized the 'schemes' management

schemes are the prefix part of uWSGI uri's. When you do

```
uwsgi --ini http://foobar.local:9090/test.ini
```

the http:// is the scheme, signalling uWSGI it has to download the config file via http.

Til now those 'schemes' were hardcoded. Now they are exposed as plugins, so you can add more of them (or override the default one).

The new system has been applied to the PSGI plugin too (sorry we are sure only perl developers will understand that kind of poetry :P) so you can do things like:

```
uwsgi --http-socket :1717 --psgi http://yourapps.local/dancer.pl
```

or

```
./uwsgi --binary-append-data yourapp.pl > blob001
cat blob001 >> ./uwsgi
./uwsgi --http-socket :1717 --psgi data://0
```

mountpoints checks

It could be hard to understand why an application server should check for mountpoints.

In the same way understanding how writing filesystem in userspace was silly few years ago.

So, check the article about managing Fuse filesystem with uWSGI: http://uwsgi-docs.readthedocs.org/en/latest/tutorials/ReliableFuse.html

Preliminary libffi plugin

As embedding c libraries for exposing hooks is becoming more common, we have started working on libffi integration, allowing safe (and sane) argument passing to hooks. More to came soon.

Official support for kFreeBSD

Debian/kFreeBSD is officially supported.

You can even use FreeBSD jails too !!!

FreeBSD Jails

Availability

uWSGI 1.9.18 has been released on October 11th 2013 and can be downloaded from:

http://projects.unbit.it/downloads/uwsgi-1.9.18.tar.gz

15.1.8 uWSGI 1.9.17

Changelog [20130917]

Bugfixes

- the 'pty' client is now blocking (safer approach)
- removed strtok() usage (substituted by a new uwsgi api function on top of strtok_r())
- fixed –pty-exec (Credits: C Anthony Risinger)
- listen_queue/somaxconn linux check is now done even for UNIX sockets

New features

The Master FIFO

This is a new management way in addition to UNIX signals. As we have no more free signals to use (and generally dealing with signals and pidfiles is not very funny), all of the new management features of uWSGI will be based on the master fifo.

Docs are already available: *The Master FIFO*

The asap hook

Credits: Matthijs Kooijman

a new hook, named 'asap' has been added. It will be run soon after the options are parsed.

Check: *Hooks*

The TCC (libtcc) plugin

TCC is an embeddable c compilers. It includes a shared library (libtcc) you can use to compile strings of c code on the fly.

The libtcc uWSGI plugins allows compiling strings of c to process symbols. CUrrently the "tcc" hook engine has been implemented:

```
[uwsgi]
hook-asap = tcc:mkdir("/var/run/sockets");printf("directory created\n");
hook-as-user = tcc:printf("i am process with pid %d\n", getpid());
hook-post-app = tcc:if (getenv("DESTROY_THE_WORLD")) exit(1);
http-socket = /var/run/sockets/foobar.sock
```

The forkptyrouter gateway

While work on Linux containers/namespaces continues to improve we have added this special router/gateway allowing dynamic allocation of pseodoterminals in uWSGI instances. To access the sockets created by the forkptyrouter you can use the –pty-connect option exposed by the 'pty' plugin.

Documention is being worked on.

added a new magic var for ANSI escaping

The %[magic var has been added, it allows you to define ANSI sequences in your logs.

If you like coloured logs:

```
log-encoder = format %[[33m${msgnl}%[[0m
```

Routable log encoders

You can now attach log encoders to specific log routes:

```
[uwsgi]
logger = stderr file:/dev/tty
log-route = stderr ubuntu
log-route = stderr clock
print = %[[34mHELLO%[[0m
; add an encoder to the 'stderr' logger
log-encoder = format:stderr %[[33m${msgnl}%[[0m
http-socket = :9090
```

–vassals-include

Credits: Matthijs Kooijman

This is like –vassal-inherit but the parsing will be "immediate" (so you can use placeholders)

The Emperor heartbeat system is now mercyless...

The old approach for the heartbeat Emperor subsystem was asking for "gentle" reload to bad vassals.

Now vassals not sending heartbeat (after being registered with the heartbeat subsystem) are killed with -9

The result of this patch will be more robust bad vassals management

logpipe

Author: INADA Naoki

You can now send loglines to the stdin of an external command:

```
req-logger = pipe:/usr/local/bin/mylogger
```

added "fd" logger to "logfile" plugin

you can directly send logs to a file descriptors:

```
req-logger = fd:17
```

Availability

uWSGI 1.9.17 has been released on Semptember 22th 2013

You can download it from:

http://projects.unbit.it/downloads/uwsgi-1.9.17.tar.gz

15.1.9 uWSGI 1.9.16

Changelog [20130914]

Important change in the gevent plugin shutdown/reload procedure !!!

The shutdown/reload phase when in gevent mode has been changed to better integrate with multithreaded (and multi-greenlet) environments (most notably the newrelic agent).

Instead of "joining" the gevent hub, a new "dummy" greenlet is spawned and "joined".

During shutdown only the greenlets spawned by uWSGI are taken in account, and after all of them are destroyed the process will exit. This is different from the old approach where the process wait for ALL the currently available greenlets (and monkeypatched threads).

If you prefer the old behaviout just specify the option –gevent-wait-for-hub

Bugfixes/Improvements

- fixed CPython reference counting bug in rpc and signal handlers
- improved smart-attach-daemon for slow processes
- follow Rack specifications for QUERY_STRING,SCRIPT_NAME,SERVER_NAME and SERVER_PORT
- report missing internal routing support (it is only a warning when libpcre is missing)
- better ipcsem support during shutdown and zerg mode (added –persistent-ipcsem as special case)
- fixed fastcgi bug exposed by apache mod_fastcgi
- do not call pre-jail hook on reload
- force linking with -lrt on solaris
- report thunder lock status
- allow custom priority in rsyslog plugin

New features

FreeBSD jails native support

uWSGI got nativr FreeBSD jails support. Official documentation is here *FreeBSD Jails*

The Rados plugin

Author: Javier Guerra

Based on the *The GlusterFS plugin* plugin, a new one allowing access to Rados object storage is available:

The RADOS plugin

The TunTap router

This new gateway is the result of tons of headaches while trying to build better (read: solid) infrastructures with Linux namespaces.

While dealing with uts, ipc, pid and filesystem namespaces is pretty handy, managing networking is a real pain.

We introduced lot of workaroud in uWSGI 1.9.15 (expecially for simplify the veth management) but finally we realized that those systems do not scale in terms of management.

The TunTap router tries to solve the issue moving the networking part of jailed vassals in user space.

Basically each vassal create one or more tuntap devices. This devices are connected (via a unix socket) to the "tuntap router" allowing access from the vassal to the external network.

That means a single network interface in the main namespace and one for each vassal.

The performance are already quite good (we are only losing about 10% in respect of kernel-level routing) but can be optimized.

In addition to this the tuntap router has a simple userspace firewall you can use to manage complex routing rules.

Documentation is still in progress, but you can configure a tuntap router following the big comment on top of this file:

https://github.com/unbit/uwsgi/blob/master/plugins/tuntap/tuntap.c

while you can connect to it with `--tuntap-device <dev> <socket>` where <dev> is the tuntap device to create in the vassal/client and <socket> is the unix address of the tuntap router

An Example Emperor

```
[uwsgi]
tuntap-router = emperor0 /tmp/tuntap.socket
exec-as-root = ifconfig emperor0 192.168.0.1 netmask 255.255.255.0 up
exec-as-root = iptables -t nat -F
exec-as-root = iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
exec-as-root = echo 1 >/proc/sys/net/ipv4/ip_forward
emperor-use-clone = ipc,uts,fs,pid,net
emperor = /etc/vassals
```

and one of its vassals:

```
[uwsgi]
tuntap-device = uwsgi0 /tmp/tuntap.socket
exec-as-root = ifconfig lo up
exec-as-root = ifconfig uwsgi0 192.168.0.2 netmask 255.255.255.0 up
exec-as-root = route add default gw 192.168.0.1
exec-as-root = hostname foobar
socket = /var/www/foobar.socket
psgi-file = foobar.pl
```

Linux O_TMPFILE

Latest Linux kernel support a new operational mode for opening files: O_TMPFILE

this flag open a temporary file (read: unlinked) without any kind of race conditions.

This mode is automatically used if available (no options needed)

Linux pivot-root

When dealing with Linux namespaces, changing the root filesystem is one of the main task.

chroot() is generally too simple, while pivot-root allows you more advanced setup

The syntax is `--pivot-root <new_root> <old_root>`

Cheaper memlimit

Author: Łukasz Mierzwa

This new check allows control of dynamic process spawning based on the RSS usage:

http://uwsgi-docs.readthedocs.org/en/latest/Cheaper.html#setting-memory-limits

Log encoders

There are dozens of log engines and storage system nowadays. The original uWSGI approach was developing a plugin for every engine.

While working with logstash and fluentd we realized that most of the logging pluging are reimplementations of teh same concept over and over again.

We followed an even more modular approach introducing log encoders:

Log encoders

They are basically patterns you can apply to each logline

New "advanced" Hooks

A new series of hooks for developers needing little modifications to the uWSGI cores are available.

Documention about the whole hooks subsystem is now available (it is a work in progress):

Hooks

New mount/umount hooks

When dealing with namespaces and jails, mounting and unmounting filesystems is one of the most common tasks.

As the mount and umount commands could not be available during the setup phase, these 2 hooks have been added directly calling the syscalls.

Check *Hooks*

Availability

uWSGI 1.9.16 has been released on September 14th 2013. You can download it from:

http://projects.unbit.it/downloads/uwsgi-1.9.16.tar.gz

15.1.10 uWSGI 1.9.15

Changelog [20130829]

Bugfixes

- fixed jvm options hashmap (#364)
- fixed python3 wsgi.file_wrapper
- fixed python3 –catch-exceptions
- fixed type in pypy wsgi.input.read
- better symbol detection for pypy
- improved ruby libraries management on heroku
- fixed http-keepalive memleak
- fixed spooler body management under CPython
- fixed unshare() usage of 'fs'
- fixed UWSGI_PROFILE usage when building plugins with –plugin
- improved SmartOS support and added OmniOS support

New features

The PTY plugin

This new plugin allows you to generate pseudoterminals and attach them to your workers.

Pseudoterminals are then reachable via network (UNIX or TCP sockets).

You can use them for shared debugging or to have input channels on your webapps.

The plugin is in early stage of development (very few features) and it is not built in by default, but you can already make funny things like:

```
[uwsgi]
plugin = pty,rack
; generate a new pseudoterminal on port 5000 and map it to the first worker
pty-socket = 127.0.0.1:5000

; classic options
master = true
processes = 2
rack = myapp.ru
socket = /tmp/uwsgi.socket

; run a ruby interactive console (will use the pseudoterminal)
```

```
; we use pry as it kick asses
rbshell = require 'pry';binding.pry
```

now you can access the pseudoterminal with

```
uwsgi --plugin pty --pty-connect 127.0.0.1:5000
```

you can run the client in various windows, it will be shared by all of the peers (all will access the same pseudoterminal).

We are sure new funny uses for it will popup pretty soon

preliminary documentation is available at *The Pty plugin*

strict mode

One of the most common error when writing uWSGI config files, are typos in option names.

As you can add any option in uWSGI config files, the system will accept anythyng you will write even if it is not a real uWSGI option.

While this approach is very powerful and allow lot of funny hacks, it can causes lot of headaches too.

If you want to check all of your options in one step, you can now add the –strict option. Unknown options will trigger a fatal error.

fallback configs

Being very cheap (in term of resources) and supporting lot of operating systems and architectures, uWSGI is heavily used in embedded systems.

One of the common feature in such devices is the "reset to factory defaults".

uWSGI now natively support this kind of operation, thanks to the –fallback-config option.

If a uWSGI instance dies with exit(1) and a fallback-config is specified, the binary will be re-exec()'d with the new config as the only argument.

Let's see an example of a configuration with unbindable address (unprivileged user trying to bind to privileged port)

```
[uwsgi]
uid = 1000
gid = 1000
socket = :80
```

and a fallback one (bind to unprivileged port 8080)

```
[uwsgi]
uid = 1000
gid = 1000
socket = :8080
```

run it (as root, as we want to drop privileges):

```
sudo uwsgi --ini wrong.ini --fallback-config right.ini
```

you will get in your logs:

```
...
bind(): Permission denied [core/socket.c line 755]
Thu Aug 29 07:26:26 2013 - !!! /Users/roberta/uwsgi/uwsgi (pid: 12833) exited with status 1 !!!
Thu Aug 29 07:26:26 2013 - !!! Fallback config to right.ini !!!
[uWSGI] getting INI configuration from right.ini
*** Starting uWSGI 1.9.15-dev-4046f76 (64bit) on [Thu Aug 29 07:26:26 2013] ***
...
```

–perl-exec and –perl-exec-post-fork

You can now run custom perl code before and after the fork() calls.

Both options simply take the perl script as the argument

uwsgi.cache_keys([cache])

This api function has been added to the python and pypy plugins. It allows you to iterate the keys of a local uWSGI cache.

It returns a list.

added *%(ftime)* to logformat

this is like 'ltime' but honouring the –log-date format

protect destruction of UNIX sockets when another instance binds them

on startup uWSGI now get the inode of the just created unix socket.

On vacuum if the inode is changed the unlink of the socket is skipped.

This should help avoiding sysadmin destructive race conditions or misconfigurations

–worker-exec2

this is like –worker-exec but happens after post_fork hooks

allow post_fork hook on general plugins

general plugins (the ones without the .request hook) can now expose the .post_fork hook

–call hooks

In the same spirit of exec-* hooks, call hooks works in the same way but directly calling functions in the current process address space (they have to be exposed as valid symbols)

take this c source (call it hello.c):

```
#include <stdio.h>

void i_am_hello_world_for_uwsgi() {
    printf("Hello World!!!\n");
}
```

and compile it as a shared library:

```
gcc -o libhello.so -shared -fPIC hello.c
```

now choose when (and where) to call it in uWSGI:

```
./uwsgi --help | grep -- --call-
 --call-pre-jail                        call the specified function before jailing
 --call-post-jail                       call the specified function after jailing
 --call-in-jail                         call the specified function in jail after initialization
 --call-as-root                         call the specified function before privileges drop
 --call-as-user                         call the specified function after privileges drop
 --call-as-user-atexit                  call the specified function before app exit and reload
 --call-pre-app                         call the specified function before app loading
 --call-post-app                        call the specified function after app loading
 --call-as-vassal                       call the specified function() before exec()ing the vassal
 --call-as-vassal1                      call the specified function(char *) before exec()ing the vass
 --call-as-vassal3                      call the specified function(char *, uid_t, gid_t) before exec
 --call-as-emperor                      call the specified function() in the emperor after the vassal
 --call-as-emperor1                     call the specified function(char *) in the emperor after the
 --call-as-emperor2                     call the specified function(char *, pid_t) in the emperor aft
 --call-as-emperor4                     call the specified function(char *, pid_t, uid_t, gid_t) in t
```

options ending with a number are variants expecting arguments (the suffix is the number of arguments they take)

we want to call our function just before our application is loaded:

```
[uwsgi]
; load the shared library
dlopen = ./libhello.so
; set the hook
call-pre-app = i_am_hello_world_for_uwsgi
...
```

your custom function will be called just before app loading.

Take in account those functions are called in the process address space, so you can make all sort of (black) magic with them.

Note: dlopen is a wrapper for the dlopen() function, so all the same rules apply (read: USE ABSOLUTE PATHS !!!)

init_func support for plugins, and –need-plugin variant

when loading a plugin you can call a symbol defined in it soon after dlopen():

```
uwsgi --plugin "foobar|myfunc" ...
```

uWSGI will call the 'myfunc' symbol exposed by the 'foobar' plugin

–need-plugin is like –plugin but will exit(1) the process if plugin loading fails

added commodity loader for the pecan framework

Author: Ryan Petrello

A new python loader (–pecan) has been added for the pecan WSGI framework

http://pecanpy.org/

https://uwsgi-docs.readthedocs.org/en/latest/Python.html#pecan-support

UWSGI_REMOVE_INCLUDES

during the build phase you can remove include headers with the UWSGI_REMOVE_INCLUDES environment variable.

This is useful for cross-compilation where some automatically detected includes could be wrong

router_expires

We already have various options in the uWSGI core to set Expires header.

This router has been added to allow customizing them:

```
[uwsgi]
route = /^foobar1(.*)/ expires:filename=foo$1poo,value=30
route = /^foobar2(.*)/ expires:unix=${time[unix]},value=30
```

the router takes a filename mtime or a unix time, adds 'value' to it, and return it as an http date.

announce Legion's death on reload/shutdown

Every legion member will now announce its death as soon as a reload (or a shutdown) of the instance is triggered

The GlusterFS plugin (beta)

This new plugin make use ot the new glusterfs c api, avoiding the overhead of fuse when serving files stored on glusterfs servers.

The plugin supports the multiprocess and multithreads modes, while async modes are currently in beta.

Documentation is available: *The GlusterFS plugin*

–force-gateway

all of the gateways (fastrouter, httprouter, rawrouter, sslrouter ...) has to be run under the master process.

By specifying –force-gateway, you will bypass this limit

preliminary python3 profiler (beta)

The –profiler pycall/pyline profilers have been added to python3. They are beta quality (they leaks memory), but should be usable.

file monitor support for OpenBSD,NetBSD,DragonFlyBSD

Both –fs-reload and the @fmon decorator now work on this operating systems.

–cwd

you can force the startup "current working directory" (used by –vacuum and the reloading subsystem) with this option.

It is useful in chroot setups where the binary executable change its place.

–add-gid

This options allows you to add additional group ids to the current process. You can specify it multiple times.

Emperor and Linux namespaces improvements

Thanks to the cooperation with the pythonanywhere.com guys the Emperor has been improved for better Linux namespaces integration.

The –emperor-use-clone option allows you to use clone() instead of fork() for your vassal's spawn. In this way you can create the vassals directly in a new namespace. The function takes the same parameters of the –unshare one

```
uwsgi --emperor /etc/vassals --emperor-use-clone pid,uts
```

will create each vassal in a new pid and uts namespace

while

```
uwsgi --emperor /etc/vassals --emperor-use-clone pid,uts,net,ipc,fs
```

will basically use all of the currently available namespaces.

Two new exec (and call) hooks are available:

–exec-as-emperor will run commands in the emperor soon after a vassal has been spawn (setting 4 env vars, UWSGI_VASSAL_CONFIG, UWSGI_VASSAL_PID, UWSGI_VASSAL_UID and UWSGI_VASSAL_GID)

–exec-as-vassal will run commands in the vassal just before calling exec() (so directly in the new namespaces)

–wait-for-interface

As dealing with the Linux network namespace introduces lot of race conditions (expecially when working with virtual ethernets), this new option allows you to pause an instance until a network interface is available.

This is useful when waiting for the emperor to move a veth to the vassal namespace, avoiding the vassal to run commands on the interface before is available

```
[uwsgi]
emperor = /etc/uwsgi/vassals
emperor-use-clone = pid,net,fs,ipc,uts
; each vassal should have its veth pair, so the following commands should be improved
exec-as-emperor = ip link del veth0
exec-as-emperor = ip link add veth0 type veth peer name veth1
; do not use the $(UWSGI_VASSAL_PID) form, otherwise the config parser will expand it on startup !!!
exec-as-emperor = ip link set veth1 netns $UWSGI_VASSAL_PID
```

```
[uwsgi]
; suspend until the emperor attach veth1
wait-for-interface = veth1
; the following hook will be run only after veth1 is available
exec-as-root = hostname vassal001
exec-as-root = ifconfig lo up
exec-as-root = ifconfig veth1 192.168.0.2
uid = vassal001
gid = vassal001
socket = :3031
...
```

Availability

uWSGI 1.9.15 has been released on August 29th 2013. You can download it from:

http://projects.unbit.it/downloads/uwsgi-1.9.15.tar.gz

15.1.11 uWSGI 1.9.14

Changelog [20130721]

Bugfixes

- fixed python modifier1 management (was hardcoded to 0)
- fixed url decoding in http and http-socket (it now supports lowercase hex, spotted by Miles Shang)
- more user-friendly error message for undeletable unix sockets
- fixed –http-auto-chunked in http 1.1 keepalive mode (André Cruz)
- fixed python wheel support (Fraser Nevett)
- fixed –safe-fd (was not correctly honoured by the Emperor)
- fixed ruby 2.x reloading
- improved support for OSX Tiger (yes, OSX 10.4)
- better computation of listen queue load
- fixed v8 build on OSX
- fixed pypy rpc
- improved chunked api performance
- fixed latin1 encoding with python3
- fixed –spooler-ordered (Roberto Leandrini)
- fixed php status line reported in request logs

New features

Ruby 1.9.x/2.x native threads support

Ruby 1.9 (mri) introduced native threads support (very similar to the CPython ones, governed by a global lock named GVL).

For various reasons (check the comments on top of the source plugin) the ruby threads support in uWSGI has been implemented as a "loop engine plugin".

You need to build the "rbthreads" plugin (it is automatic when using the 'ruby2' build profile) and enable it with '–rbthreads'

The gem script has been extended, automatically selecting the 'ruby2' build profile when a ruby >= 1.9 is detected (this should make the life easier for Heroku users)

Rails4 is the first Ruby on Rails version supporting and blessing threads (in 3.x you need to explicitely enable support). You can use multiple threads in Rails4 only when in "production" mode, otherwise your app will deadlock after the first request.

An example config:

```
[uwsgi]
plugins = rack,rbthreads
master = true
; spawn 2 processes
processes = 2
; spawn 8 threads
threads = 8
; enable ruby threads
rbthreads = true
; load the Rack app
rack = config.ru
; bind to an http port
http-socket = :9090
http-socket-modifier1 = 7
```

it will generate a total of 16 threads

Filesystem monitoring interface (fsmon)

Currently uWSGI is able to monitor filesystem changes using the "simple" –touch-* facility or the signal framework (using various operating system api like inotify or kqueue).

A new interface for plugin writers named "fsmon" has been added, allowing easy implementation of realtime filesystem monitors.

Three new options have been added:

–fs-reload <path>

–fs-brutal-reload <path>

–fs-signal <path> <signal>

Contrary to the –touch-* options they are realtime (the master is woke up as soon as the item changes) and. uses kernel facilities (currently only inotify() and kqueue() are supported). Thanks to this choice you can now monitor a whole directory for changes (without the need of external processes/wrapper like inotifywatch)

uClibc support

Author: Natanael Copa

uWSGI can now be built on uclibc-based systems (generally, embedded systems)

Alpine Linux is the operating system on which the support has been tested

Lua 5.2 support

Author: Natanael Copa

the lua plugins now supports Lua 5.2

setscheme, setdocroot

This two new routing actions allow you to dinamically override DOCUMENT_ROOT and UWSGI_SCHEME

sendfile, fastfile

This two actions (added to the router_static plugin) allows you to return static files to the client bypassing the DOCUMENT_ROOT check.

The first one forces the use of the sendfile() syscall (where available), while the second automatically tries to choose the best serving strategy (like offloading)

–reload-on-fd and –brutal-reload-on-fd

Two new options allowing you to reload an instance when a file descriptor is ready.

Currently the best usage scenario is for the oom_control cgroup interface (via eventfd).

Supposing you have a process wrapper allocating an eventfd() reporting OOM events (and exposed as the 'OOM' environment var) you can force a uWSGI reload when out of memory with:

```
[uwsgi]
...
reload-on-fd = $(OOM):8 OUT OF MEMORY !!!
```

it means:

monitor the $(OOM) file descriptor and read 8 bytes from it when ready (it is an eventfd() requirement), then print "OUT OF MEMORY !!!" in the logs and gracefully reload the instance.

Obviously this is only a way to use it. The UNIX world is file-descriptor based so you have plenty of funny ways to use it.

Spooler improvements

Author: Roberto Leandrini

Effectively all of the work has been done in uwsgidecorators.py

You can now pass to all of the available spooler-related decorators the "pass_arguments=True" option, to automatically serialize the spooler function parameters. This is an abstraction avoiding you the need to serialize/deserialize arguments.

In addition to this the decorators have been extended to implement __call__ in this way you can directly call spoller decorated functions as normal functions.

–emperor-nofollow

Enabling this option will allows the Emperor to watch for symbolic links mtime update instead of the mtime of the real file.

Alberto Scotto is working on an updated version supporting both (should be ready for the next release)

daemontools envdir support

Albeit daemontools look old-fashioned, things like envdirs (http://cr.yp.to/daemontools/envdir.html) are heavily used in various context.

uWSGI got two new options (–envdir <path> and –early-envdir <path>) allowing you to support this special (arcaic ?) configuration way.

xmldir improvements

Author: Guido Berhoerster

The xmldir plugins has been improved supporting iconv-based utf8 encoding. Various minor fixes have been committed.

The examples directory containes two new files showing an xmldir+xslt usage

Breaking News !!!

Servlet 2.5 support development has just started. The plugin is present in the tree but it is unusable (it is an hardcoded jsp engine). We expect a beta version after the summer. Obviously we shameless consider *The JWSGI interface* a better approach than servlet for non-Enterprise people ;)

Availability

Download uWSGI 1.9.14 from

http://projects.unbit.it/downloads/uwsgi-1.9.14.tar.gz

15.1.12 uWSGI 1.9.13

Changelog [20130622]

Bugfixes

- Fixed a corner case bug when response offloading is enabled, but no request plugin is loaded
- Fixed harakiri routing when multiple rules are in place (return NEXT instead of CONTINUE)
- Fixed curl crashing master on slow dns responses (Łukasz Mierzwa)
- Removed PTRACE check in uwsgi.h (it is no more needed since uWSGI 1.0)
- Fixed –print-sym
- Added a newline in –cflags
- Improved python3 detection and compilation
- Fixed Coro::AnyEvent loop engine (John Berthels)
- Rack api functions are now static
- Better fastcgi handling of big uploads
- Improved GCC usage on Darwin for Python non-apple builds
- Fixed XCLIENT usage in rawrouter
- Use the clang preprocessor instead of hardcoded 'cpp' when CC=clang is used
- Set 16bit options to 65535 when higher values are requested
- Fixed virtualhosting (it is now compatible with 1.4 configurations)

New features

PyPy performance and features improvents

The PyPy plugin has been improved a lot. The amount of C code has been reduced by 70%, so, now, the vast majority of the plugin is written in python. The c helpers have been removed allowing the python part to directly call native uWSGI functions via cffi.

Support for PyPy continulets (and their greenlet abstraction) has been added (while waiting for a solid gevent port for pypy) and a chat example is already available (using the uwsgi async api):

https://github.com/unbit/uwsgi/tree/master/t/pypy

https://github.com/unbit/uwsgi/blob/master/contrib/pypy/uwsgi_pypy_greenlets.py

The pypy uwsgi api has been improved and now you can use the uwsgidecorators module (even if the spooler subsystem is still missing)

Chunked input api

In the last days there have been a bunch of discussions on how to correctly manage chunked input. As basically none of the available standards support it in a "definitive" way, we have defined a low-level api (so we can easily adapt it in the feature).

The api exposes two functions:

uwsgi.chunked_read()

and

uwsgi.chunked_read_nb()

A non blocking chat example:

```
import uwsgi
def application(e, sr):
    while True:
        uwsgi.wait_fd_read(uwsgi.connection_fd())
        uwsgi.suspend()
        msg = uwsgi.chunked_read_nb()
        if msg: print "core %d" % e['uwsgi.core'], msg
```

Toward better third-party plugins management: the –dot-h option

As the –cflags option shows the CFLAGS used to build the server, the –dot-h option shows the content of uwsgi.h

This means the content of uwsgi.h is now embedded in the binary (compressed where available).

It could look a bizarre choice but the objective is to allow easy compilation of plugins out of the uwsgi sources (somethign that will be available in 2.0 for sure)

setmethod, seturi and setpathinfo routing action

we continue extending the routing api.

Three new actions have been added to dinamically modify the request

UWSGI_INCLUDES

You can now ovverride the include search path (while building uWSGI) with this environment variable.

Improved set_user_harakiri api function

Now the uwsgi.set_user_harakiri automatically recognize mules and spoolers. It has been added to the ruby/rack, pypy and perl/psgi plugins

–add-cache-item [cache]KEY=VALUE

this is a commodity option (mainly useful for testing) allowing you to store an item in a uWSGI cache during startup

the router_xmldir plugin

This is a proof of concept plugin aimed at stressing the transformation api.

It basically generates an xml representation of a directory. This could be useful to implement apache-style directoryindex:

Check this example using xslt:

https://github.com/unbit/uwsgi/issues/271#issuecomment-19820204

Implement __call__ for @spool* decorators

Thanks to 'anaconda', you can now directly call functions mapped to the spooler, so instead of

```
myfunc.spool(args)
```

you can directly do:

```
myfunc(args)
```

the old way is obviously supported

the uwsgi[lq] routing var

this routing var exports the current size of the listen_queue:

```
[uwsgi]
...
route-if = higher:${uwsgi[lq]};70 break:503 Server Overload
...
```

–use-abort

On some system the SEGV signal handler cannot be correctly restored after the uWSGI backtrace.

If you want to generate a core file, you may want to trigger a SIGABRT soon after the backtrace.

Availability

uWSGI 1.9.13 will be available 22th June 2013 at this url:

http://projects.unbit.it/downloads/uwsgi-1.9.13.tar.gz

15.1.13 uWSGI 1.9.12

Changelog [20130605]

Bugfixes

- offloading cache writes will return the correct status code and not 202
- you can now control the path of temporary files setting the TMPDIR environment variable (this fixes an old issue for users without control over /tmp)
- fixed a compilation error on amqp imperial monitor
- cron commands are correctly escaped when reported in the stats server
- fixed fastcgi parser corner-case bug with big uploads
- fixed support for newest cygwin

New Features

Offloading responses

Take the following WSGI app:

```
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return ['u' * 100000000]
```

it will generate about 100megs of data. 98% of the time the worker spent on the request was on the data transfer. As the whole response is followed by the end of the request we can offload the data write to a thread and free the worker suddenly (so it will be able to handle a new request).

100megs are a huge value, but even 1MB can cause a dozen of poll()/write() syscalls that blocks your worker for a bunch of milliseconds

Thanks to the 'memory offload' facility added in 1.9.11 implementing it has been very easy.

The offloading happens via the *uWSGI Transformations*

```
[uwsgi]
socket = :3031
wsgi-file = myapp.py
; offload all of the application writes
route-run = offload:
```

By default the response is buffered to memory until it reaches 1MB size. After that it will be buffered to disk and the offload engine will use sendfile().

You can set the limit (in bytes) after disk buffering passing an argument to the offload:

```
[uwsgi]
socket = :3031
wsgi-file = myapp.py
; offload all of the application writes (buffer to disk after 1k)
route-run = offload:1024
```

"offload" MUST BE the last transformation in the chain

```
[uwsgi]
socket = :3031
wsgi-file = myapp.py
; gzip the response
route-run = gzip:
; offload all of the application writes (buffer to disk after 1k)
route-run = offload:1024
```

JWSGI and JVM improvements

The JVM plugin has been extended to support more objects helper (like ArrayList), while JWSGI can now be used as a low-level layer to add support for more JVM-based languages.

JRuby integration is the first attempt of such a usage. We have just releases a JWSGI to Rack adapter allowing you tun run Ruby/Rack apps on top of JRUBY:

https://github.com/unbit/jwsgi-rack

A similar approach for Jython is on work

–touch-signal

A new touch option has been added allowing the rise of a uwsgi signal when a file is touched:

```
[uwsgi]
...
; raise signal 17 on /tmp/foobar modifications
touch-signal = /tmp/foobar 17
...
```

The "pipe" offload engine

A new offload engine allowing transfer from a socket to the client has been added.

it will be automatically used in the new router_memacached and router_redis plugins

memcached router improvements

You can now store responses in memcached (as you can already do with uWSGI caching)

```
[uwsgi]
...
route = ^/cacheme memcachedstore:addr=127.0.0.1:11211,key=${REQUEST_URI}
route = ^/cacheme2 memcachedstore:addr=192.168.0.1:11211,key=${REQUEST_URI}foobar
...
```

obviously you can get them too

```
[uwsgi]
...
route-run = memcached:addr=127.0.0.1:11211,key=${REQUEST_URI}
...
```

The memcached router is now builtin in the default profiles

The new redis router

Based on the memcached router, a redis router has been added. It works in the same way:

```
[uwsgi]
...
route = ^/cacheme redisstore:addr=127.0.0.1:6379,key=${REQUEST_URI}
route = ^/cacheme2 redisstore:addr=192.168.0.1:6379,key=${REQUEST_URI}foobar
...
```

... and get the values

```
[uwsgi]
...
route-run = redis:addr=127.0.0.1:6379,key=${REQUEST_URI}
...
```

The redis router is builtin by default

The "hash" router

this special routing action allows you to hash a string and return a value from a list (indexed with the hashed key).

Take the following list:

127.0.0.1:11211

192.168.0.1:11222

192.168.0.2:22122

192.168.0.4:11321

and a string:

/foobar

we hash the string /foobar using djb33x algorithm and we apply the modulo 4 (the size of the items list) to the result.

We get "1", so we will get the second items in the list (we are obviously zero-indexed).

Do you recognize the pattern ?

Yes, it is the standard way to distribute items on multiple servers (memcached clients for example uses it from ages).

The hash router exposes this system allowing you to distribute items in you redis/memcached servers or to make other funny things.

This an example usage for redis:

```
[uwsgi]
...
; hash the list of servers and return the value in the MYNODE var
route = ^/cacheme_as/(.*) hash:items=127.0.0.1:11211;192.168.0.1:11222;192.168.0.2:22122;192.168.0.4
; log the result
route = ^/cacheme_as/(.*) log:${MYNODE} is the choosen memcached server !!!
; use MYNODE as the server address
route = ^/cacheme_as/(.*) memcached:addr=${MYNODE},key=$1
...
```

you can even choose the hashing algo from those supported in uWSGI

```
[uwsgi]
...
; hash the list of servers with murmur2 and return the value in the MYNODE var
route = ^/cacheme_as/(.*) hash:algo=murmur2,items=127.0.0.1:11211;192.168.0.1:11222;192.168.0.2:22122
; log the result
route = ^/cacheme_as/(.*) log:${MYNODE} is the choosen memcached server !!!
; use MYNODE as the server address
route = ^/cacheme_as/(.*) memcached:addr=${MYNODE},key=$1
...
```

the router_hash plugin is compiled-in by default

Availability

uWSGI 1.9.12 will be available starting from 20130605 at the following url

http://projects.unbit.it/downloads/uwsgi-1.9.12.tar.gz

15.1.14 uWSGI 1.9.11

Changelog [20130526]

Bugfixes

- Fixed Python 3 stdout/stderr buffering
- Fixed mule messages (`@mulefunc` is now reliable)
- Fixed `SCRIPT_NAME` handling in dynamic mode
- Fixed X-Sendfile with gzip static mode
- Fixed cache item maximum size with custom block size
- Fixed cache path handling

New features

The new high-performance PyPy plugin

Credits: Maciej Fijalkowski

We are pleased to announce the availability of the new PyPy plugin.

PyPy team has been great in helping us. We hope the uWSGI integration (that exposed new challenges to the PyPy project) will help PyPy becaming better and better.

Official docs: *The PyPy plugin*

Cron improvements

Credits: Łukasz Mierzwa

Unique crons You can now avoid overlapping crons. The uWSGI master will track death of a single task, and until its death the same cron will not be triggered:

```
[uwsgi]
unique-cron = -1 -1 -1 -1 -1 my_script.sh
```

cron2 syntax A key/value variant of the –cron option is now available:

```
[uwsgi]
cron2 = minute=39,hour=23,month=-1,week=-1,day=-1,unique=1,legion=foobar,harakiri=30
```

harakiri cron When using the `cron2` option you are allowed to set a harakiri timeout for a cron task. Just add `harakiri=n` to the options.

Support for GNU Hurd

Debian GNU/Hurd has been recently released. uWSGI 1.9.11 can be built over it, however very few tests have been made.

The memory offload engine

Idea: Stefano Brentegani

When serving content from the cache, a worker could get blocked during transfer from memory to the socket.

A new offload engine named "memory" allows to offload memory transfers. The cache router automatically supports it. Support for more areas will be added soon.

To enable it just add `--offload-threads <n>`

New Websockets chat example

An example websocket chat using Redis has been added to the repository:

https://github.com/unbit/uwsgi/blob/master/tests/websockets_chat.py

Error routes

You can now define a routing table to be executed as soon as you set the HTTP status code in your plugin.

This allows you to completely modify the response. This is useful for custom error codes.

All of the routing standard options are available (included labels) plus an optimized `error-route-status` matching a specific HTTP status code:

```
[uwsgi]
error-route-status = 502 redirect:http://unbit.it
```

Support for corner case usage in wsgi.file_wrapper

Generally the `wsgi.file_wrapper` callable expects a file-like object. PEP 333/3333 reports a special pattern when the object is not a file (call `read()` until the object is consumed). uWSGI now supports this pattern (even if in a hacky way).

HTTP/HTTPS router keepalive improvements

Credits: André Cruz

When using `--http-keepalive` you can now hold the connection open even if the request has a body.

The harakiri routing action

You can now set a harakiri timer for each request using internal routing:

```
[uwsgi]
; set harakiri to 30 seconds for request starting with /slow
route = ^/slow harakiri:30
```

RPC wrappers

The RPC plugin has been extended to allows interoperation with other standards.

Currently a simple HTTP wrapper and an XML-RPC one are exposed.

The HTTP simple wrapper works by parsing `PATH_INFO`.

A `/foo/bar/test` call will result in

uwsgi.rpc('foo', 'bar', 'test')

To enable this HTTP mode just set the `modifier2` to '2':

```
[uwsgi]
http-socket = :9090
http-socket-modifier1 = 173
http-socket-modifier2 = 2
; load the rpc code
import = myrpcfuncs.py
```

or (to have more control)

```
[uwsgi]
http-socket = :9090
route-run = uwsgi:,173,2
; load the rpc code
import = myrpcfuncs.py
```

The XML-RPC wrapper works in the same way, but it uses the modifier2 value '3'. It requires a libxml2-enabled build of uWSGI.

```
[uwsgi]
http-socket = :9090
route-run = uwsgi:,173,3
; load the rpc code
import = myrpcfuncs.py
```

Then just call it:

```
proxy = xmlrpclib.ServerProxy("http://localhost:9090')
proxy.hello('foo','bar','test')
```

You can combine multiple wrappers using routing.

```
[uwsgi]
http-socket = :9090
; /xml force xmlrpc wrapper
route = ^/xml uwsgi:,173,3
; fallback to HTTP simple
route-if-not = startswith:${PATH_INFO};/xml uwsgi:,173,2
; load the rpc code
import = myrpcfuncs.py
```

Availability

uWSGI 1.9.11 will be available since 20130526 at:

http://projects.unbit.it/downloads/uwsgi-1.9.11.tar.gz

15.1.15 uWSGI 1.9.10

Changelog [20130511]

Bugfixes

- fixed alarm threads during reloads
- fixed uninitialized memory in –touch-* options
- fixed a regression in –attach-daemon

New Features

Welcome to gccgo

Go support in gcc 4.8 is amazing, thanks to the split-stack feature you can now have goroutines without allocating a whole pthread.

As Go 1.1 will be no no more compatible with uWSGI, gccgo will became the official way for running go apps.

The gccgo plugin is in early stage of development but it is already able to run in preforking mode.

We are heavy working on a true "goroutines" Loop engine. Stay tuned.

Final routes

You can now run routing rules after a request. Obviously not all of the exposed actions make sense after the request but you should be able to write even more complex setup.

Check this request limiter based on HTTP response status (a value you can get only after a request):

https://github.com/unbit/uwsgi/blob/master/t/routing/errorlimiter.ini

Availability

uWSGI 1.9.10 will be available since 20130511 at the following url:

http://projects.unbit.it/downloads/uwsgi-1.9.10.tar.gz

15.1.16 uWSGI 1.9.9

Changelog [20130508]

Special Warning !!!

The router_basicauth plugin has changed its default behaviour to return "break" if authorization fails.

The "basicauth-next" action, uses the old behaviour (returning "next")

This new approach should reduce security problems caused by wrong configurations

Bugfixes

- do not increment "tx" statistics counter for "unaccountable" plugins
- fixed –backtrace-depth
- fixed cache-sync parsing
- fixed mule farms initialization
- fixed multithreading bug when regexp conditional route is used
- fixed default-app usage in the psgi plugin
- fixed python dynamic mode + threads
- fixed error reporting in corerouter when retry is in place
- correctly report harakiri condition for gateways

New Features

The WebDav plugin

WebDav is one of the much requested features for the project. We now have a beta-quality plugin, already supporting additional standards like the carddav:

https://github.com/unbit/uwsgi/blob/master/t/webdav/carddav.ini

The official modifier is 35, and to mount a simple directory as a webdav shares (for use with windows, gnome...) you only need to specify the –webdav-mount option:

```
[uwsgi]
plugin = webdav
http-socket = :9090
http-socket-modifier1 = 35
webdav-mount = /home/foobar
```

remember to protect shares:

```
[uwsgi]
plugin = webdav,router_basicauth
http-socket = :9090
http-socket-modifier1 = 35
route-run = basicauth:CardDav uWSGI server,unbit:unbit
webdav-mount = /home/foobar
```

WebDav attributes are stored as filesystem xattr, so be sure to use a filesystem supporting them (ext4, xfs, hfs+...)

LOCK/UNLOCK support is still incomplete

Official docs will be available soon.

Support for Go 1.1 (more or less, sad news for go users...)

Albeit you can successfully embed go 1.1 apps in uWSGI, go 1.1 will be completely fork() unsafe.

That means you are not able to use multiprocessing, the master, mules and so on.

Basically half of the uWSGI features will be no more usable in go apps.

Things could change in the future, but currently our objective is better integration with the gccgo project.

Go 1.0.x will continue to be supported (unless gccgo shows itself as a better alternative)

More to come soon.

Improved async modes

Stackless, Greenlet and Fiber support have been updated to support new async features

The radius plugin

You can now authenticate over radius servers using the router_radius plugin:

```
[uwsgi]
plugin = webdav,router_radius
http-socket = :9090
http-socket-modifier1 = 35
route-run = radius:realm=CardDav uWSGI server,server=127.0.0.1:1812
webdav-mount = /home/foobar
```

The SPNEGO plugin

Another authentication backend, using SPNEGO (kerberos)

```
[uwsgi]
plugin = webdav,router_spnego
http-socket = :9090
http-socket-modifier1 = 35
route-run = spnego:HTTP@localhost
webdav-mount = /home/foobar
```

The plugin is beta quality as it leaks memory (it looks like a bug in MIT-kerberos) and Heimdal implementation does not work.

More reports are wellcomed

The ldap authenticator

(Author: Łukasz Mierzwa)

Currently it lacks SASL support. Will be improved soon.

```
[uwsgi]
...
plugins = router_ldapauth
route = ^/a ldapauth:LDAP realm,url=ldap://ldap.domain,com;basedn=ou=users,dc=domain.com;binddn=uid=p
```

New internal routing features

We removed the GOON action, as it was messy and basically useless with the new authentication approach

The “setscriptname” action has been added to override the internally computed SCRIPT_NAME (not only the var)

The "donotlog" action forces uWSGI to not log the current request

The "regexp" routing conditions has been improved to allows grouping. Now you can easily manipulate strings and adding them as new request VARS:

```
[uwsgi]
...
route-if = regexp:${REQUEST_URI};^/(.)oo addvar:PIPPO=$1
route-run = log:PIPPO IS ${PIPPO}
```

this will take the first char of foo and place in the PIPPO request var

Gevent atexit hook

uwsgi.atexit hook is now honoured by the gevent plugin (Author: André Cruz)

Streaming transformations

Transformations can be applied on the fly (no buffering involved).

Check updated docs: *uWSGI Transformations*

The xattr plugin

The xattr plugin allows you to reference files extended attributes in the internal routing subsystem:

```
[uwsgi]
...
route-run = addvar:MYATTR=user.uwsgi.foo.bar
route-run = log:The attribute is ${xattr[/tmp/foo:MYATTR]}
```

or (variant with 2 vars)

```
[uwsgi]
...
route-run = addvar:MYFILE=/tmp/foo
route-run = addvar:MYATTR=user.uwsgi.foo.bar
route-run = log:The attribute is ${xattr2[MYFILE:MYATTR]}
```

The airbrake plugin

(Author: Łukasz Mierzwa)

Currently at early stage of development allows sending uWSGI exceptions and alarms to airbrake servers.

Official docs will be available soon.

Legion Daemons

(Author: Łukasz Mierzwa)

No, it is not a blackmetal band, it is a new feature of *The uWSGI Legion subsystem* allowing you to run external processes only when an instance is a lord:

```
[uwsgi]

master = true
http = :8081
stats = :2101
wsgi-file = tests/staticfile.py

logdate = true

legion = legion1 225.1.1.1:19678 100 bf-cbc:abc
legion-node = legion1 225.1.1.1:19678

legion-attach-daemon = legion1 memcached -p 10001

legion-smart-attach-daemon = legion1 /tmp/memcached.pid memcached -p 10002 -d -P /tmp/memcached.pid
```

–touch-exec

A new "touch" option (like –touch-reload) is available, triggering the execution of a command:

```
[uwsgi]
...
touch-exec = /tmp/foobar run_my_script.sh
touch-exec = /var/test/foo.txt run_my_second_script.sh arg1 arg2
```

Math for cache

You can now use the caching subsystem to store 64bit signed numbers and apply atomic operations on them.

The uwsgi api has been extended with 5 new functions (currently exposed only by the python plugin):

*uwsgi.cache_num(key[,cache]) -> get the 64bit number from the specified item

*uwsgi.cache_inc(key[,amount=1,expires,cache]) -> increment the specified key by the specified amount

*uwsgi.cache_dec(key[,amount=1,expires,cache]) -> deccrement the specified key by the specified amount

*uwsgi.cache_mul(key[,amount=2,expires,cache]) -> multiply the specified key by the specified amount

*uwsgi.cache_div(key[,amount=2,expires,cache]) -> divide the specified key by the specified amount

The new api has been exposed to the routing subsystem, allowing you to implement advanced patterns, like the request limiter:

https://github.com/unbit/uwsgi/blob/master/t/routing/limiter.ini

the example shows hot to limit the request of a single ip to 10 every 30 seconds

The long-term objective of this new feature is being the base for the upcoming metric subsystem

Availability

uWSGI 1.9.9 will be availabel since 20130508 at the following url

http://projects.unbit.it/downloads/uwsgi-1.9.9.tar.gz

15.1.17 uWSGI 1.9.8

Changelog [20130423]

Note: this is an "emergency" release fixing 2 regressions causing a crash during reloads and when using async+uGreen

Bugfixes

- fixed a crash when reloading the master
- fixed a crash in async mode + uGreen
- the 'mime' routing var requires a request var (not a raw string)

Availability

You can download uWSGi 1.9.8 from http://projects.unbit.it/downloads/uwsgi-1.9.8.tar.gz

15.1.18 uWSGI 1.9.7

Bugfixes

- fixed teajs engine build
- fixed offloading status code (set to 202 when a request is offloaded)
- execute cron tasks within 60 second resolution, instead of 61 seconds
- fixed websocket proxy
- check for python3 unicode encoding (instead of crashing...)
- fixed ipcsem removal on reload
- fixed kqueue timer on OpenBSD, NetBSD and DragonFlyBSD
- fixed/reimplemented perl uwsgi::register_rpc
- fixed fd leak on sendfile() error
- fixed Content-Length when gzip file variant is used
- allows non-request plugins to register rpc functions
- more robust error checking for cgroups
- honour SCRIPT_NAME the in the PSGI plugin when multiple perl apps are mounted

New features

Legion cron

A common needs when multiple instances of an application are running, is to force only one of them to run cron tasks. The new –legion-cron uses *The uWSGI Legion subsystem* to accomplish that:

```
[uwsgi]
; use the new legion-mcast shortcut (with a valor 90)
legion-mcast = mylegion 225.1.1.1:9191 90 bf-cbc:mysecret
; run the script only if the instance is the lord of the legion "mylegion"
legion-cron = mylegion -1 -1 -1 -1 -1 my_script.sh
```

Curl cron

The curl_cron plugin has been added allowing the cron subsystem to call urls (via libcurl) instead of unix commands:

```
[uwsgi]
; call http://uwsgi.it every minute
curl-cron = -1 -1 -1 -1 -1 http://uwsgi.it/
```

The output of the request is reported in the log

The UWSGI_EMBED_PLUGINS build variable

ou can now embed plugins on the fly during the build phase. Check this example:

```
UWSGI_EMBED_PLUGINS=gridfs,rack UWSGI_PROFILE=psgi make
```

this will build a monolithic binary with the default profile for psgi + the gridfs and the rack plugins (both embedded in the binary)

Gzip caching

The cachestore routing function can now directly store items in gzip format.

Check the CachingCookbook: http://uwsgi-docs.readthedocs.org/en/latest/tutorials/CachingCookbook.html

–skip-atexit

A bug in the mongodb client library could cause a crash of the uWSGI server during shutdown/reload. This option avoid calling atexit() hooks. If you are building a *The GridFS plugin* infrastructure you may want to use this option while the MongoDB guys solve the issue.

proxyhttp and proxyuwsgi

The http and uwsgi routing instructions are now more smart. You can cache their output and get the right status code in the logs.

This requires you to NOT use offloading. If offloading is in place and do not want to use it for this two router use the proxy-prefixed variant that will skip offloading.

You can now make cool things like:

```
[uwsgi]
socket = 127.0.0.1:3031
; create a cache of 100 items
cache = 100
; check if a cached value is available
```

```
route-run = cache:key=${REQUEST_URI}
; proxy all request to http://unbit.it
route-run = http:81.174.68.52:80,unbit.it
; and cache them for 5 minutes
route-run = cachestore:key=${REQUEST_URI},expires=300
```

The transformation api

A generic api for manipulating the response has been added (cachestore uses it)

check *uWSGI Transformations*

–alarm-fd

We are improving *The uWSGI alarm subsystem (from 1.3)* to be less-dependent on loglines. You can now trigger alarms when an fd is ready for read.

This is really useful for integration with the Linux eventfd() facility.

For example you can monitor (and throw an alarm) when your cgroup is running the OOM-Killer:

```
[uwsgi]
; define an 'outofmemory' alarm that simply print the alarm in the logs
alarm = outofmemory log:
; raise the alarm (with the specified message) when fd is ready (this is an eventfd se we read 8 byte
alarm-fd = outofmemory $(CGROUP_OOM_FD):8 OUT OF MEMORY !!!
```

in this example CGROUP_OOM_FD is an environment variable mapping to the number of an eventfd() filedescriptor inherited from some kind of startup script. Maybe (in the near future) we could be able to directly define this kind of monitor directly in uWSGI.

More information on the eventfd() + cgroup integration are here: https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

an example perl startup script:

```
use Linux::FD;
use POSIX;

my $foo = Linux::FD::Event->new(0);
open OOM,'/sys/fs/cgroup/uwsgi/memory.oom_control';
# we dup() the file as Linux::FD::Event set the CLOSE_ON_EXEC bit (why ???)
$ENV{'CGROUP_OOM_FD'} = dup(fileno($foo)).'';

open CONTROL,'>/sys/fs/cgroup/uwsgi/cgroup.event_control';
print CONTROL fileno($foo).' '.fileno(OOM)."\n";
close CONTROL;

exec 'uwsgi','mem.ini';
```

The spooler server plugin and the cheaper busyness algorithm compiled in by default

In extremely high-loaded scenario the busyness cheaper algorithm (by Łukasz Mierzwa) has been a real silver bullet in the past months allowing adaptive process spawning to be based on real usage time taking in account performance and response time. For this reason the plugin is now builtin by default.

In addition to this the remote spooler plugin (allowing external process to enqueue jobs) has been added too in the default build profile.

Availability

uWSGI 1.9.7 will be available since 20130422 at this url:

http://projects.unbit.it/downloads/uwsgi-1.9.7.tar.gz

15.1.19 uWSGI 1.9.6

Changelog 20130409

Bugfixes

- workaround for building the python plugin with gcc 4.8

Sorry, this is not a real bugfix, but making a release without bugfixes seems wrong...

New Features

Sqlite and LDAP pluginization

Storing configurations in sqlite databases or LDAP tree is a pretty "uncommon" way to configure uWSGI instances. For such a reason they have been moved to dedicated plugins.

If you store config in a sqlite database, just add –plugin sqlite3. For LDAP, just add –plugin ldap:

```
uwsgi --plugin sqlite --sqlite config.db
```

Configuring dynamic apps with internal routing

'Til now, you need to configure your webserver to load apps dinamically.

Three new instructions have been added to load aplication on demand.

Check the example:

```
[uwsgi]

http-socket = :9090

route = ^/foo chdir:/tmp
route = ^/foo log:SCRIPT_NAME=${SCRIPT_NAME}
route = ^/foo log:URI=${REQUEST_URI}
route = ^/foo sethome:/var/uwsgi/venv001
route = ^/foo setfile:/var/uwsgi/app001.py
route = ^/foo break:

route = ^/bar chdir:/var
route = ^/bar addvar:SCRIPT_NAME=/bar
route = ^/bar sethome:/var/uwsgi/venv002
route = ^/bar setfile:/var/uwsgi/app002.py
route = ^/bar break:
```

as you can see, rewriting SCRIPT_NAME is now very easy. The sethome instruction is currently available only for python application (it means 'virtualenv')

Carbon avg computation (Author: Łukasz Mierzwa)

You can now configure how the carbon plugin send the response average when no requests have been managed.

You have three ways:

–carbon-idle-avg none - don't push any avg_rt value if no requests were made

—carbon-idle-avg last - use last computed avg_rt value (default)

–carbon-idle-avg zero - push 0 if no requests were made

Numeric checks for the internal routing

New check are available:

ishigher or '>'

islower or '<'

ishigherequal or '>='

islowerequal or '<='

Example:

```
[uwsgi]
route-if = ishigher:${CONTENT_LENGTH};1000 break:403 Forbidden
```

Math and time for the internal routing subsystem

If you build uWSGI with matheval support (matheval-dev on debian/ubuntu) you will get math support in your routing system via the 'math' routing var.

The 'time' routing var has been added currently exporting only the 'unix' field returning the epoch.

Check this crazy example:

```
[uwsgi]
http-socket = :9090
route-run = addvar:TEMPO=${time[unix]}
route-run = log:inizio = ${TEMPO}
route-run = addvar:TEMPO=${math[TEMPO+1]}
route-run = log:tempo = ${TEMPO}
```

As you can see the routing subsystem can store values in request variables (here we create a 'TEMPO' var, and you will be able to access it even in your app request vars)

The 'math' operations can reference request vars

Check the matheval docs for the supported operations: http://matheval.sourceforge.net/docs/index.htm

Added non-standard seek() and tell() to wsgi.input (post-buffering required)

While testing the 'smart mode' of the 'Klaus' project (https://github.com/jonashaag/klaus) we noticed it was violating the WSGI standard calling seek() and tell() when in smart mode.

We have added support for both methods when post-buffering is enabled.

REMEMBER: they violate the WSGI standard, so try to avoid them (if you can). There are better ways to accomplish that.

Pyshell improvements, AKA Welcome IPython (Idea: C Anthony Risinger)

You can invoke the ipython shell instead of the default one when using –pyshell:

```
uwsgi -s :3031 --pyshell="from IPython import embed; embed()"
```

Obviously you can pass whatever code to –pyshell

The 'rpcraw' routing instruction

Another powerful and extremely dangerous routing action. It will call a rpc function sending its return value directly to the client (without further processing).

Empty return values means "go to the next routing rule".

Return values must be valid HTTP:

```
uwsgi.register_rpc('myrules', function(uri) {
     if (uri == '/foo') {
             return "HTTP/1.0 200 OK\r\nContent-Type: text/plain\r\nServer: uWSGI\r\nFoo: bar\r\n\r\n
     }
     return "";
});
```

```
[uwsgi]
plugin = v8
v8-load = rules.js
route = ^/foo rpcraw:myrules ${REQUEST_URI}
```

Preliminary support for the HTTP Range header

The range request header allows requesting only part of a resource (like a limited set of bytes of a static file).

The system can be used when serving static files, but it is disabled by default. Just add –honour-range to enable it.

In the future it will be used for file wrappers (like wsgi.file_wrapper) and for *The GridFS plugin* (this is the reason for not enabling it by default as you could have already implemented range management in your app)

The 'lord' routing condition

We are working hard on making a truly amazing cluster subsystem using *The uWSGI Legion subsystem*

You can now execute internal routing rules when an instance is a lord:

```
[uwsgi]
...
route-if = lord:mylegion log:I AM THE LORD !!!
```

the "I AM THE LORD !!!" logline will be printed only when the instance is a lord of the legion 'mylegion'

GridFS authentication

You can now connect to authenticated MongoDB servers when using *The GridFS plugin*

Just add the username and password parameters to the mount definition

The –for-times config logic

You can use –for-times for running uWSGI options the specified number of times:

```
[uwsgi]
for-times = 8
   mule = true
endfor =
```

this will spawn 8 mules

The 'uwsgi' routing var

Accessing uWSGI internal parameters when defining routing rules could be handy. The 'uwsgi' routing var is the container for such vars.

Currently it exports 'wid' (the id of the worker running the rule) and 'pid' (the pid of the worker running the rule)

```
[uwsgi]
master = true
processes = 4
; stupid rule... break connections to the worker 4
route-if = ishigher:${uwsgi[wid]};3 break:403 Forbidden
```

The 'alarm' routing action

You can now trigger alarms from the routing subsystem:

```
[uwsgi]

alarm = pippo cmd:cat

route = ^/help alarm:pippo ${uwsgi[wid]} ${uwsgi[pid]}
http-socket = :9090
```

when /help is requested the 'pippo' alarm is triggered passing the wid and the pid as the message

Welcome to the ruby shell

As well as the –pyshell we now have the ruby shell:

```
uwsgi --rbshell -s :3031
```

or

```
uwsgi --rbshell="require 'pry';binding.pry" -s :3031
```

for using the pry shell: http://pryrepl.org/

... and welcome to the Lua shell

As python and ruby, even Lua got its shell. Just add –lua-shell

Goodbye to the old (and useless) probe subsystem

The probe subsystem was added during 0.9 development cycle but it was badly designed and basically broken.

It has been definitely removed (the deprecation phase has been skipped as 1.9 is not an LTS release and 1.4 still support it)

Improvements in the Legion subsystem (Author: Łukasz Mierzwa)

Two new hooks have been added: –legion-node-joined and –legion-node-left

More fine-tuning

–socket-sndbuf and –socket-rcvbuf have been added to allow tuning of the send a receive buffer of the uWSGI sockets (use with caution)

V8 improvements and TeaJS integration

The *uWSGI V8 support* plugin continue to improve. The main target is still *uWSGI internal routing* but JSGI support is almost complete and we are working for TeaJS (old v8cgi) integration: http://code.google.com/p/teajs/

more to come soon...

Availability

uWSGI 1.9.6 will be available since 20130409 at this url:

http://projects.unbit.it/downloads/uwsgi-1.9.6.tar.gz

15.1.20 uWSGI 1.9.5

Changelog 20130404

Bugfixes

- fixed a memory leak with cachestore routing instruction (Riccardo Magliocchetti)
- fixed a memory leak in carbon plugin (Riccardo Magliocchetti)
- fixed a memory leak in the cgi plugin (Riccardo Magliocchetti)
- fixed old-style python dynamic apps
- force the emperor to honour –max-fd for vassals
- improved PSGI seek with post-buffering
- fixed kvlist escaping

New features

The GridFS plugin

A plugin exporting GridFS features is available, check official docs: *The GridFS plugin*

V8 improvements

The V8 plugin continues to improve. Preliminary JSGI 3.0 support is available as well as multithreading.

The 'require' commonjs standard has been implemented.

Writing commonjs specs will be a very long work, so maybe a partnership with projects like teajs (the old v8cgi) would be a better path to follow.

In the mean time, we are working on docs: *uWSGI V8 support*

The 'cgi' routing instruction

You can now call CGI script directly from the *uWSGI internal routing*

```
[uwsgi]
plugin = cgi
route = ^/cgi-bin/(.+) cgi:/usr/lib/cgi-bin/$1
```

Availability

uWSGI 1.9.5 will be available since 20130404 at this url

http://projects.unbit.it/downloads/uwsgi-1.9.5.tar.gz

15.1.21 uWSGI 1.9.4

Changelog 20130330

Bugfixes

fixed cache statistics exported by the stats subsystem (Łukasz Mierzwa)

fixed CoroEV bug in after_request (Tom Molesworth and John Berthels)

update cache items after a restore from persistent storage (Łukasz Mierzwa)

fixed signal handling in non-worker processes

fixed thundering herd in multiple mules setup

ported the cplusplus skeletal plugin to the new api

fixed uWSGI reloading when build as a shared library

New features

SmartOS official support

From now on, SmartOS is included in the officially supported operating systems

V8 initial support

The Lua previous suggestion for writing uWSGI routing rules and configurations, woke up lot of javascript users stating that javascript itself could be a valid alternative. A V8 plugin is now available, supporting RPC, signal handlers and configurations. You need libv8 headers to build it:

```
python uwsgiconfig.py --plugin plugins/v8
```

```
var config = {};
config['socket'] = [':3031', ':3032', ':3033'];
config['master'] = true;
config['processes'] = 3+1;
config['module'] = 'werkzeug.testapp:test_app';

config;
```

```
uwsgi --plugin v8 --config foo.js
```

The previous example will allows you to write dynamic configs in javascript, while you can export javascript functions via the RPC subsystem:

```
function part1(request_uri, remote_addr) {
     return '<h1>i am part1 for ' + request_uri + ' ' + remote_addr + "</h1>" ;
}

function part2(request_uri, remote_addr) {
     return '<h2>i am part2 for ' + request_uri + ' ' + remote_addr + "</h2>" ;
}

function part3(request_uri, remote_addr) {
     return '<h3>i am part3 for ' + request_uri + ' ' + remote_addr + "</h3>" ;
}

uwsgi_register_rpc('part1', part1);
uwsgi_register_rpc('part2', part2);
uwsgi_register_rpc('part3', part3);
```

```
[uwsgi]
plugin = v8
v8-load = func.js
cache2 = name=foobar,items=10

http-socket = :9090

route-run = addheader:Content-Type: text/html
route-run = cache:key=pippo,name=foobar
route-run = cachestore:key=pippo,name=foobar
route-run = rpcnext:part1 ${REQUEST_URI} ${REMOTE_ADDR}
route-run = rpcnext:part2 ${REQUEST_URI} ${REMOTE_ADDR}
route-run = rpcnext:part3 ${REQUEST_URI} ${REMOTE_ADDR}
route-run = break:
```

The previous example generates an HTTP reponse from 3 javascript functions and store it in the uWSGI cache.

Curious about rpcnext ?

The rpcnext routing action

We can already call rpc functions from the routing subsystem to generate response. With the rpcnext action (aliased as rpcblob too) you can call multiple rpc functions and assemble the return values in a single response.

Legion improvements

We are hardly working in stabilizing *The uWSGI Legion subsystem* The objective is have a rock-solid clustering implementation for uWSGI 2.0 that you can use even from your applications.

The code in 1.9.4 has been refactored a bit by Łukasz Mierzwa to allow easier integration with external plugins.

A new "join" hook has been added, it is called as soon as a node becomes active part of a legion (read, it is part of a quorum).

Availability

uWSGI 1.9.4 will be available since 20130330 at this url

http://projects.unbit.it/downloads/uwsgi-1.9.4.tar.gz

15.1.22 uWSGI 1.9.3

Changelog 20130328

Bugfixes

fixed imports in the JVM build system when virtualenvs are used (Ryan Kaskel)

fixed mod_proxy_uwsgi with apache 2.4

fixed php headers generation when Status is created from the php app itself

New features

Pluggable configuration system (with Lua support)

From this version you will be able to implement configurators (like the already available xml, ini, yaml, json, ldap, sqlite...) as plugins.

The first available configurator is the Lua one (offered by the lua plugin).

This is an example configuration written in lua:

```
config = {}

config['immediate-uid'] = 'roberto'
config['immediate-gid'] = 'roberto'
config['http-socket'] = ':9090'
config['env'] = { 'FOO=bar', 'TEST=topogigio' }
config['module'] = 'werkzeug.testapp:test_app'

return config
```

you can load it with:

```
uwsgi --plugin lua --config config.lua
```

The –config option is the way to load pluggable configurators. You can even override the already available embedded configurators with your own version.

The Emperor has already been extended to support pluggable configurators:

```
[uwsgi]
emperor = /etc/uwsgi/vassals
emperor-extra-extension = .lua
emperor-extra-extension = .foo
```

adding emperor-extra-extension will allows the emperor to search for the specified extension passing the config file to the vassal with the –config option.

Immediate setuid and setgid

In a recent uWSGI maling-list thread, the need to not rely on file system permissions for the tyrant mode emerged.

Albeit it is the most secure approach, two new options –immediate-uid and –immediate-gid have been added.

Setting them on top of your vassal file will force the instance to setuid()/setgid() as soon as possibile and without the (theoretical) possibility to override them.

The word “theoretical” here is the key, you always need to remember that a security bug in uWSGI could allow a malicious user to change privileges, so if you really care security (or do not trust uWSGI developers ;) always drop privileges before the vassal/instance is spawned (like in standard tyrant mode)

Honouring symlinks in tyrant mode

The option –emperor-tyrant-nofollow has been added forcing the emperor to now follow symlinks when searching for uid/gid in tyrant mode.

This option allows the sysadmin to simply symlink configurations and just change the uid/gid of the symlink it self (remember to pass the -h option to chown !!!)

The "rpcret" routing action (or usa Lua to write advanced rules)

The *uWSGI internal routing* continue to be improved.

You can already call rpc function for the routing system (to generate response bypassing WSGI/PSGI/Rack/... engines):

```
[uwsgi]
lua-load = myrpcfunctions.lua
route = ^/foo/(.+)/call rpc:hello_world ${REMOTE_ADDR} $1
```

the hello_worls rpc function is defined (and registered) in the myrpcfunctions.lua taking two arguments.

The function is called when the routing regexp matches, and its output sent to the client.

The "rpcret" works in similar way, but instead generating a response, you generate a routing return code:

```
function choose(request_uri, remote_addr)
   print( 'REQUEST_URI is ' ..request_uri.. ' (from Lua)')
   if request_uri == '/topogigio' then
     return "goto topogigio"
   end
   return "break 500 Internal server Error !!!"
end

print('Hello Hello')
uwsgi.register_rpc('choose', choose)
```

and the uWSGI config:

```
[uwsgi]
route-run = rpcret:choose ${REQUEST_URI} ${REMOTE_ADDR}
route-run = break

route-label = topogigio
route-run = log:i am topogigio !!!
```

The 'choose' rpc function will be invoked at every request passing REQUEST_URI and REMOTE_ADDR as its argument.

The return string of the function will be used to know what to do next (from the internal ruting point of view).

Currently supported return strings are:

`next` move to the next rule

`continue` pass the request to the request handler

`goon` move to the next rule with a different action

`break` close the connection with an optional status code

`goto <label>` goto to the specified label

Obviously rpc functions for rpcret can be written in any language/platform supported by uWSGI, but we strongly suggest to go with Lua for performance reasons (the inpact compared to pure C code is pretty irrelevant). If you are lucky and can use LuaJit you will experiment even better performance as for this kind of job a JIT compiler is the best approach.

Availability

uWSGI 1.9.3 has been released on 20130328 and can be downloaded from:

http://projects.unbit.it/downloads/uwsgi-1.9.3.tar.gz

15.1.23 uWSGI 1.9.2

Changelog 20130326

Bugfixes

Fixed python3 response headers managament (wrong refcnt)

Fixed readline() on request body when postbuffering is in place

Fixed ruby fiber plugin

New features

route-run and the cachestore routing action

You can now store responses automatically in the uWSGI cache:

```
[uwsgi]
http-socket = :9090
; ensure the sweeper thread will run
master = true
cache2 = name=pippo2,items=10
module = werkzeug.testapp:test_app
route-run = cache:key=${REQUEST_URI},name=pippo2
route-run = cachestore:key=${REQUEST_URI},expires=30,name=pippo2
```

this example check every request for its availability in the cache 'pippo2'. If not available the request plugin (werkzeug test app) will run normally and its output will be stored in the cache (only if it returns a HTTP 200 status)

`--route-run` is a new option allowing you to directly call routing action without checking for a specific condition (yes, it is an optimization)

routing access to cookie and query string

Check updated docs *uWSGI internal routing*

empty internal routing condition

Check updated docs *uWSGI internal routing*

The Geoip plugin

Check official docs *The GeoIP plugin*

The SSI plugin (beta)

Check official docs *SSI (Server Side Includes) plugin*

Availability

uWSGI 1.9.2 has been released on 20130326 and can be downloaded from:

http://projects.unbit.it/downloads/uwsgi-1.9.2.tar.gz

15.1.24 uWSGI 1.9.1

First minor release for the 1.9 tree.

Bugfixes

Fixed –req-logger after a graceful reload

Fixed a crash with the carbon plugin

Fixed signal handling when multiple workers + copy on write is in place

Fixed exception handling in the Rack plugin

The XSLT plugin

The *XSLT* plugin has been added. It allows to apply XML transformation via request plugin or *uWSGI internal routing*

Legion scrolls api

Scrolls are text blob attached to each member of a *Legion* cluster. We are slowly defining an api allowing developers to directly use the legion subsystem in their apps and configurations. The addition in 1.9.1 is the uwsgi.scrolls(legion) function returning a list/array of the current scrolls defined by the whole cluster. This is still not something fully usable (and useful) more to come soon...

On demand vassals

Another step in better resource usage for massive hosting. You can now tell the *Emperor* to start vassals only after the first request to a specific socket. Combined with –idle/–die-on-idle options, you can have truly on-demand applications.

To define the socket to wait for for each vassal you have 3 options:

–emperor-on-demand-extension <ext>

this will instruct the Emperor to check for a file named <vassal>+<ext>, if the file is available it will be read and its content used as the socket to wait for:

```
uwsgi --emperor /etc/uwsgi/vassals --emperor-on-demand-extension .socket
```

supposing a myapp.ini file in /etc/uwsgi/vassals, a /etc/uwsgi/vassals/myapp.ini.socket will be searched for (and its content used as the socket name)

At the first connection, the vassal is spawned and the socket passed as the file descriptor 0. File descriptor 0 is always checked by uWSGI so you do not need to specify a –socket option in the vassal file. This works automagically for uwsgi sockets, if you use other protocols (like http or fastcgi) you have to specify it with the –protocol option

–emperor-on-demand-directory <dir>

This is a less-versatile approach supporting only UNIX sockets. Basically the name (without extension and path) of the vassal is appended to the specified directory + the .socket extension and used as the on-demand socket:

```
uwsgi --emperor /etc/uwsgi/vassals --emperor-on-demand-directory /var/tmp
```

using the previous example, the socket /var/tmp/myapp.socket will be automatically bound

–emperor-on-demand-exec <cmd>

This is what (very probably) you will use in very big deployments. Every time a new vassal is added the supplied command is run passing the vassal name as the first argument. The STDOUT of the command is used as the socket name.

The –exec-post-app hook

In addition to the other –exec-* options (used to run commands at the various server stages), a new one has been added allowing you to run commands after the load of an application.

The pyring build profile

This is a very specific build profile allowing you to automatically build a uWSGI stack with monolithic python support and modular jvm + ring honouring virtualenvs.

The cache router plugin

This has been improved, and in next releases we should be able to directly store response in the uWSGI cache only using the internal routing subsystem

Docs will be available soon

The crypto logger

If you host your applications on cloud services without persistent storage you may want to send your logs to external systems. Sadly logs often contain sensible informations you should not transfer in clear. The new crypto logger try to solve this issue allowing you to encrypt each log packet and send it over udp to a server able to decrypt it.

The following example

```
uwsgi --plugin logcrypto --logger crypto:addr=192.168.173.22:1717,algo=bf-cbc,secret=ciaociao -M -p 4
```

will send each log packet to the udp server available at 192.168.173.22:1717 encrypting the text with ‘ciaociao’ secret key using the blowfish cbc algorithm.

An example server is available here:

https://github.com/unbit/uwsgi/blob/master/contrib/cryptologger.rb

The rpc internal routing instruction

The "rpc" routing instruction has been added, allowing you to call rpc functions directly from the routing subsystem and forward they output to the client.

Check the following examples:

```
[uwsgi]
http-socket = :9090
route = ^/foo addheader:Content-Type: text/html
route = ^/foo rpc:hello ${REQUEST_URI} ${HTTP_USER_AGENT}
route = ^/bar/(.+)$ rpc:test $1 ${REMOTE_ADDR} uWSGI %V
route = ^/pippo/(.+)$ rpc:test@127.0.0.1:4141 $1 ${REMOTE_ADDR} uWSGI %V
import = funcs.py
```

Preliminary support for name resolving in the carbon plugin

You can specify carbon servers using hostnames. The current code is pretty simple. Future updates will support round robin queries.

New routing conditions

New routing conditions have been added (equal,startswith,endswith,regexp) check the updated docs:

http://uwsgi-docs.readthedocs.org/en/latest/InternalRouting.html#the-internal-routing-table

The 'V' magic var

You can reference the uWSGI version string using the %V magic var in your configurations

The 'mongodb' generic plugin

This is a commodity plugin for packagers not able to access a shared libmongoclient. This basically link it in a new shared object that can be used by the others mongodb plugin

Build profiles over network

You can now reference build profiles using urls (http, https and ftp are supported):

```
UWSGI_PROFILE=http://uwsgi.it/psgi.ini make
```

Get it

uWSGI 1.9.1 will be available since 20130324 at this url:

http://projects.unbit.it/downloads/uwsgi-1.9.1.tar.gz

15.1.25 uWSGI 1.9

This is the version that will lead to the LTS 2.0. It includes a lot of internal changes and removal of a lot of basically unused, broken, or too ugly functionality.

Options deprecated in 1.0.x have been definitely removed.

Non-blocking for all

From now on, all of the request plugins, need to be non-blocking. A new set of C/C++/Obj-C api have been added to help the user/developer writing non-blocking code in a safe way. Plugins like the RPC one have been rewritten using that new api, allowing you to use it with engines like Gevent or Coro::Anyevent The async mode has been rewritten to better cooperate with this new rule. More info can be found on *uWSGI asynchronous/non-blocking modes (updated to uWSGI 1.9)* The new async mode requires some form of coroutine/greenthread/suspend engine to correctly work. Again, check *uWSGI asynchronous/non-blocking modes (updated to uWSGI 1.9)*

Coro::AnyEvent

The Perl/PSGI plugin is one of the most ancient in the uWSGI project, but used to not support the async mode in advanced ways.

Thanks to the new *uWSGI asynchronous/non-blocking modes (updated to uWSGI 1.9)* mode, a Coro::Anyevent (coroae) loop engine has been added.

To build it you need the Coro::Anyevent package (you can use cpanm to get it), then just add –coroae <n> to your options where <n> is the number of async cores to spawn.

The JVM plugin

We finally have a truly working JVM infrastructure in uWSGI 1.9. Check the new docs at *JVM in the uWSGI server (updated to 1.9)* Improved *The JWSGI interface* support is available as well as the new Clojure *The Clojure/Ring JVM request handler* plugin

The Mono ASP.NET plugin

The first Mono plugin attempt (in 2010) was a total failure. Now we have a new shining implementation.

Check docs here *The Mono ASP.NET plugin*

Language independent HTTP body management

One of the most annoying task in writing uWSGI request plugins, was re-implementing the management of HTTP body reader every time.

The new non-blocking api added 3 simple generic C/C++/Obj-C functions to deal with it in a language independent way:

```
char *uwsgi_request_body_read(struct wsgi_request *wsgi_req, ssize_t hint, ssize_t *rlen);
char *uwsgi_request_body_readline(struct wsgi_request *wsgi_req, ssize_t hint, ssize_t *rlen);
void uwsgi_request_body_seek(struct wsgi_request *wsgi_req, off_t pos);
```

they automatically manage post-buffering, non-blocking and upload progress.

All of the request plugins have been updated to the new api

Faster uwsgi/HTTP/FastCGI/SCGI native sockets

All of the –socket protocol parsers have been rewritten to be faster (less syscall usage) and to use less memory. They are now more complex, but you should note (on loaded site) a reduced amount of syscalls per-request.

The SCGI protocol support has been added, while a NPH fastcgi mode (where the output is HTTP instead of cgi) has been implemented.

The FastCGI protocol now supports true sendfile() usage

The old behaviour of storing the request body for HTTP and FastCGI on a temp file, has been removed (unless you use post-buffering). This means you can now have upload progress with protocols other than uwsgi.

Request logging VS err logging

One of the most annoying problem with older uWSGI releases was the lack of ability to easily split request logs from error logs. You can now create a logger and map it only to request logging:

```
[uwsgi]
req-logger = syslog
...
```

As an example you may want to send request logging to syslog and redis, and error log to mongodb (on the foo.bar collection):

```
[uwsgi]
req-logger = syslog
req-logger = redislog:127.0.0.1:6269
logger = mongodblog:127.0.0.1:9090,foo.bar
...
```

Or just use (boring) files

```
[uwsgi]
req-logger = file:/tmp/reqlog
logger = file:/tmp/errlog
...
```

Chain reloading

When in lazy/lazy_apps mode, you can simply destroy a worker to force it to reload the application code.

A new reloading system named “chain reload”, allows you to reload one worker at time (opposed to the standard way where all of the workers are destroyed in bulk)

Chain reloading can only be triggered via “touch”: –touch-chain-reload <file>

Offloading improvements

Offloading appeared in uWSGI 1.4 and is one of the most loved features. In 1.9 we added a new engine: “write”, that allows you to offload the write of files on disk. A general function api uwsgi.offload() is on work, to allow applications to access the offload engine. All of the uWSGI parts sending static files (including the language-specific implementations, like WSGI wsgi.file_wrapper) have been extended to automatically use offloading if available. This means you can use your Framework’s way for serving static files, without losing too much performance and (more important) without blocking your workers.

Better static files management/serving

uWSGI 1.9 received many improvements in static file serving.

You may want to check: *Serving static files with uWSGI (updated to 1.9)*

For syadmins one of the most interesting new features is the ability to use the uWSGI new generation cacheing (see below) to store request -> absolute_path mappings

The New Generation Cache subsystem (cache2)

The uWSGI caching subsystem has been completely rewritten to be a more general purpose in-memory key/value store. The old caching subsystem has been re-built on top of it, and is now more of a general "web caching" system. The new cache subsystem allows you to control all of the aspects of your memory store, from the hashing algorithm to the amount of blocks.

You can now have multiple caches per-instance (identified by name)

To create a cache just use the new –cache2 option

```
[uwsgi]
cache2 = name=mycache,items=100
cache2 = name=faster,items=200,hash=murmur2,keysize=100,blocksize=4096
cache2 = name=fslike,items=1000,keysize=256,bitmap=1,blocks=2000,blocksize=8192
...
```

In this example we created 3 caches: mycache, faster and fslike.

The first one is a standard old-style, cache able to store 100 items of a maximum size of 64k with keys limited to 2048 bytes using djb33x hashing algorithm The second one use the murmur2 hashing algorithm, each key can be at most 1000 bytes, can store 200 items of max 4k The last one works like a filesystem, where each item can span over multiple blocks. That means, fslike cache can save lot of memory for boject of different size (but it will be slower than non-bitmap based caches)

The options you can specify in cache2 are the following:

`name` the name of the cache (must be unique) REQUIRED

`items/max_items/maxitems` set the max number of items the cache can store REQUIRED

`blocksize` set the size of a single block

`blocks` set the number of blocks (used only in bitmap mode)

`hash` set the hashing algorithm, currently supported: djbx33 and murmur2

`hashsize/hash_size` set the size of the hash table (default to 65536 items)

`keysize/key_size` set the max size of a key

`store` set the filename in which to persistent store the cache

`store_sync/storesync` set the frequency (in seconds) at which msync() is called to flush cache on disk (when in persistent mode)

`node/nodes` the new cache subsystem can send cache updates via udp packet. With this option you set one or more (separated with ;) udp addresses on which to send updates

`sync` set it to the address of a cache server. Its whole content will be copied in the new cache (use it for initial sync)

`udp/udp_servers/udp_server/udpservers/udpserver` bind to the specified udp addresses (separated with ;) listening for cache updates

`bitmap` enable botmap mode (set it to 1)

If you are asking yourself why such low-level tunings exists, you have to take in account that the new caching subsystem is used in lot of areas, so for different needs you may want different tuning. Just check *Scaling SSL connections (uWSGI 1.9)* for an example

The old –cache-server option has been removed. The threaded cache server added in 0.9.8 has been completed superseeded by the new non blocking infrastructure. If you load the "cache" plugin (enabled by default in monolithic build) a cache server will be available and managed by the workers.

Update docs are available here *The uWSGI caching framework*

The Legion subsystem

The Legion subsystem is a new whole addition to the uWSGI project. It has superseeded the old Clustering subsystem (which has been removed in 1.9). It implements a quorum system to manage shared resources in clustered environments. Docs are already available: *The uWSGI Legion subsystem*

Cygwin (windows) support

uWSGI can be compiled on windows machines using the cygwin POSIX emulation system. The event subsystem uses simple poll() (mapped to select() on cygwin), while the lock engine uses windows mutexes. Albeit from our tests it looks pretty solid, we consider the porting still "experimental"

Advanced Exceptions subsystem

As well as the request body language-independent management, an exception management system has been added. Currently supported only in the Python and Ruby plugins, allows language-independent handling of exceptions cases (like reloading on a specific exception). The –catch-exception option has been improved to show lot of useful information. Just try it (in development !!!) Future development will allow automatic sending of exception to system like Sentry or Airbrake.

SPDY, SSL and SNI

Exciting new features have been added to the SSL system and the HTTP router

SPDY support (currently only version 3) will get lot of users attention, but SNI subsystem is what sysadmins will love

Preliminary docs are available

The SPDY router (uWSGI 1.9)

SNI - Server Name Identification (virtual hosting for SSL nodes)

HTTP router keepalive, auto-chunking, auto-gzip and transparent websockets

Many users have started using the HTTP/HTTPS/SPDY router in production, so we started adding features to it. Remember this is ONLY a router/proxy, NO I/O is allowed, so you may not be able to throw away your old-good webserver.

The new options:

`--http-keepalive` enable HTTP/1.1 keepalive connections

`--http-auto-chunked` for backend response without content-length (or chunked encoding already enabled), transform the output in chunked mode to maintain keepalive connections

`--http-auto-gzip` automatically gzip content if uWSGI-Encoding header is set to gzip, but content size (Content-Length/Transfer-Encoding) and Content-Encoding are not specified

`--http-websockets` automatically detect websockets connections to put the request handler in raw mode

The SSL router (sslrouter)

A new corerouter has been added, it works in the same way as the rawrouter one, but will terminate ssl connections. The sslrouter can use sni for implementing virtualhosting (using the –sslrouter-sni option)

Websockets api

20Tab S.r.l. (a company working on HTML5 browsers game) sponsored the development of a fast language-independent websockets api for uWSGI. The api is currently in very good shape (and maybe faster than any other implementation). Docs still need to be completed but you may want to check the following examples (a simple echo):

https://github.com/unbit/uwsgi/blob/master/tests/websockets_echo.pl (perl)

https://github.com/unbit/uwsgi/blob/master/tests/websockets_echo.py (python)

https://github.com/unbit/uwsgi/blob/master/tests/websockets_echo.ru (ruby)

New Internal Routing (turing complete ?)

The internal routing subsystem has been rewritten to be 'programmable'. You can see it as an apache mod_rewrite with steroids (and goto ;) Docs still need to be ported, but the new system allows you to modify/filter CGI vars and HTTP headers on the fly, as well as managing HTTP authentication and caching.

Updated docs here (still work in progress) *uWSGI internal routing*

Emperor ZMQ plugin

A new imperial monitor has been added allowing vassals to be governed over zeromq messages:

http://uwsgi-docs.readthedocs.org/en/latest/ImperialMonitors.html#zmq-zeromq

Total introspection via the stats server

The stats server now exports all of the request variables of the currently running requests for each core, so it works in multithread mode too. This is a great way to inspect what your instance is doing and how it does it In the future, uwsgitop could be extended to show the currently running request in realtime.

Nagios plugin

Ping requests sent using nagios plugin will no longer be counted in apps request stats. This means that if application had –idle option enabled nagios pings will no longer prevent app from going to idle state, so starting with 1.9 –idle should be disabled when nagios plugin is used. Otherwise app may be put in idle state just before nagios ping request, when ping arrives it needs to wake from idle and this might take longer than ping timeout, causing nagios alerts.

Removed and deprecated features

- The –app option has been removed. To load applications on specific mountpoints use the –mount option
- The –static-offload-to-thread option has been removed. Use the more versatile –offload-threads
- The grunt mode has been removed. To accomplish the same behaviour just use threads or directly call fork() and uwsgi.disconnect()
- The send_message/recv_message api has been removed (use language-supplied functions)

Working On, Issues and regressions

We missed the timeline for a bunch of expected features:

- SPNEGO support, this is an internal routing instruction to implement SPNEGO authentication support
- Ruby 1.9 fibers support has been rewritten, but need tests
- Erlang support did not got required attention, very probably will be post-poned to 2.0
- Async sleep api is incomplete
- SPDY push is still not implemented
- RADIUS and LDAP internal routing instructions are unimplemented
- The channel subsystem (required for easy websockets communications) is still unimplemented

In addition to this we have issues that will be resolved in upcoming minor releases:

- the –lazy mode lost usefulness, now it is like –lazy-apps but with workers-reload only policy on SIGHUP
- it looks like the JVM does not cooperate well with coroutine engines, maybe we should add a check for it
- Solaris and Solaris-like systems did not get heavy testing

Special thanks

A number of users/developers helped during the 1.9 development cycle. We would like to make special thanks to:

Łukasz Mierzwa (fastrouters scalability tests)

Guido Berhoerster (making the internal routing the new skynet)

Riccardo Magliocchetti (static analysis)

André Cruz (HTTPS and gevent battle tests)

Mingli Yuan (Clojure/Ring support and test suite)

15.2 LTS releases

15.2.1 uWSGI 1.4.10 (LTS)

Bugfixes

- fixed python3 static files handling (via wsgi.file_wrapper)
- backported python3 latin1 fix from 1.9

- fixed –backtrace-depth
- fixed python3 pyargv
- fixed mule_msg pipe handling

Availability

uWSGI 1.4.10 has been released 20130823

You can download it from:

http://projects.unbit.it/downloads/uwsgi-1.4.10.tar.gz

CHAPTER 16

Contact

Mailing list	http://lists.unbit.it/cgi-bin/mailman/listinfo/uwsgi
Gmane mirror	http://dir.gmane.org/gmane.comp.python.wsgi.uwsgi.general
IRC	#uwsgi @ irc.freenode.org. The owner of the channel is *unbit*.
Twitter	http://twitter.com/unbit
Commercial support	http://unbit.com/

.

CHAPTER 17

Commercial support

You can buy commercial support from http://unbit.com

CHAPTER 18

Donate

uWSGI development is sponsored by the Italian ISP Unbit and its customers. You can buy commercial support and licensing. If you are not an Unbit customer, or you cannot/do not want to buy a commercial uWSGI license, consider making a donation. Obviously please feel free to ask for new features in your donation.

We will give credit to everyone who wants to sponsor new features.

See the old uWSGI site for the donation link. You can also donate via GitTip.

CHAPTER 19

Indices and tables

- *genindex*
- *modindex*
- *search*

u

uwsgi, 385
uwsgidecorators, 395