CherryPy Documentation

CherryPy Team

Jan 17, 2021

CONTENTS

CHAPTER

ONE

FOREWORD

1.1 Why CherryPy?

CherryPy is among the oldest web framework available for Python, yet many people aren't aware of its existence. One of the reason for this is that CherryPy is not a complete stack with built-in support for a multi-tier architecture. It doesn't provide frontend utilities nor will it tell you how to speak with your storage. Instead, CherryPy's take is to let the developer make those decisions. This is a contrasting position compared to other well-known frameworks.

CherryPy has a clean interface and does its best to stay out of your way whilst providing a reliable scaffolding for you to build from.

Typical use-cases for CherryPy go from regular web application with user frontends (think blogging, CMS, portals, ecommerce) to web-services only.

Here are some reasons you would want to choose CherryPy:

1. Simplicity

   Developing with CherryPy is a simple task. "Hello, world" is only a few lines long, and does not require the developer to learn the entire (albeit very manageable) framework all at once. The framework is very pythonic; that is, it follows Python's conventions very nicely (code is sparse and clean).

   Contrast this with J2EE and Python's most popular and visible web frameworks: Django, Zope, Pylons, and Turbogears. In all of them, the learning curve is massive. In these frameworks, "Hello, world" requires the programmer to set up a large scaffold which spans multiple files and to type a lot of boilerplate code. CherryPy succeeds because it does not include the bloat of other frameworks, allowing the programmer to write their web application quickly while still maintaining a high level of organization and scalability.

   CherryPy is also very modular. The core is fast and clean, and extension features are easy to write and plug in using code or the elegant config system. The primary components (server, engine, request, response, etc.) are all extendable (even replaceable) and well-managed.

   In short, CherryPy empowers the developer to work with the framework, not against or around it.

2. Power

   CherryPy leverages all of the power of Python. Python is a dynamic language which allows for rapid development of applications. Python also has an extensive built-in API which simplifies web app development. Even more extensive, however, are the third-party libraries available for Python. These range from object-relational mappers to form libraries, to an automatic Python optimizer, a Windows exe generator, imaging libraries, email support, HTML templating engines, etc. CherryPy applications are just like regular Python applications. CherryPy does not stand in your way if you want to use these brilliant tools.

   CherryPy also provides *tools* and *plugins*, which are powerful extension points needed to develop world-class web applications.

3. Maturity

   Maturity is extremely important when developing a real-world application. Unlike many other web frameworks, CherryPy has had many final, stable releases. It is fully bugtested, optimized, and proven reliable for real-world use. The API will not suddenly change and break backwards compatibility, so your applications are assured to continue working even through subsequent updates in the current version series.

   CherryPy is also a "3.0" project: the first edition of CherryPy set the tone, the second edition made it work, and the third edition makes it beautiful. Each version built on lessons learned from the previous, bringing the developer a superior tool for the job.

4. Community

   CherryPy has an devoted community that develops deployed CherryPy applications and are willing and ready to assist you on the CherryPy mailing list or Gitter. The developers also frequent the list and often answer questions and implement features requested by the end-users.

5. Deployability

   Unlike many other Python web frameworks, there are cost-effective ways to deploy your CherryPy application.

   Out of the box, CherryPy includes its own production-ready HTTP server to host your application. CherryPy can also be deployed on any WSGI-compliant gateway (a technology for interfacing numerous types of web servers): mod_wsgi, FastCGI, SCGI, IIS, uwsgi, tornado, etc. Reverse proxying is also a common and easy way to set it up.

   In addition, CherryPy is pure-python and is compatible with Python 2.3. This means that CherryPy will run on all major platforms that Python will run on (Windows, MacOSX, Linux, BSD, etc).

   webfaction.com, run by the inventor of CherryPy, is a commercial web host that offers CherryPy hosting packages (in addition to several others).

6. It's free!

   All of CherryPy is licensed under the open-source BSD license, which means CherryPy can be used commercially for ZERO cost.

7. Where to go from here?

   Check out the *tutorials* to start enjoying the fun!

1.2 Success Stories

You are interested in CherryPy but you would like to hear more from people using it, or simply check out products or application running it.

If you would like to have your CherryPy powered website or product listed here, contact us via our mailing list or Gitter.

1.2.1 Websites running atop CherryPy

Hulu Deejay and Hulu Sod - Hulu uses CherryPy for some projects. "The service needs to be very high performance. Python, together with CherryPy, gunicorn, and gevent more than provides for this."

Netflix - Netflix uses CherryPy as a building block in their infrastructure: "Restful APIs to large applications with requests, providing web interfaces with CherryPy and Bottle, and crunching data with scipy."

Urbanility - French website for local neighbourhood assets in Rennes, France.

MROP Supply - Webshop for industrial equipment, developed using CherryPy 3.2.2 utilizing Python 3.2, with libs: Jinja2-2.6, davispuh-MySQL-for-Python-3-3403794, pyenchant-1.6.5 (for search spelling). "I'm coming over from .net development and found Python and CherryPy to be surprisingly minimalistic. No unnecessary overhead - build everything you need without the extra fluff. I'm a fan!"

CherryMusic - A music streaming server written in python: Stream your own music collection to all your devices! CherryMusic is open source.

YouGov Global - International market research firm, conducts millions of surveys on CherryPy yearly.

Aculab Cloud - Voice and fax applications on the cloud. A simple telephony API for Python, C#, C++, VB, etc... The website and all front-end and back-end web services are built with CherryPy, fronted by nginx (just handling the ssh and reverse-proxy), and running on AWS in two regions.

Learnit Training - Dutch website for an IT, Management and Communication training company. Built on CherryPy 3.2.0 and Python 2.7.3, with oursql and DBUtils libraries, amongst others.

Linstic - Sticky Notes in your browser (with linking).

Almad's Homepage - Simple homepage with blog.

Fight.Watch - Twitch.tv web portal for fighting games. Built on CherryPy 3.3.0 and Python 2.7.3 with Jinja 2.7.2 and SQLAlchemy 0.9.4.

1.2.2 Products based on CherryPy

SABnzbd - Open Source Binary Newsreader written in Python.

Headphones - Third-party add-on for SABnzbd.

SickBeard - "Sick Beard is a PVR for newsgroup users (with limited torrent support). It watches for new episodes of your favorite shows and when they are posted it downloads them, sorts and renames them, and optionally generates metadata for them."

TurboGears - The rapid web development megaframework. Turbogears 1.x used Cherrypy. "CherryPy is the underlying application server for TurboGears. It is responsible for taking the requests from the userâ€™s browser, parses them and turns them into calls into the Python code of the web application. Its role is similar to application servers used in other programming languages".

Indigo - "An intelligent home control server that integrates home control hardware modules to provide control of your home. Indigo's built-in Web server and client/server architecture give you control and access to your home remotely from other Macs, PCs, internet tablets, PDAs, and mobile phones."

SlikiWiki - Wiki built on CherryPy and featuring WikiWords, automatic backlinking, site map generation, full text search, locking for concurrent edits, RSS feed embedding, per page access control lists, and page formatting using PyTextile markup."

read4me - read4me is a Python feed-reading web service.

Firebird QA tools - Firebird QA tools are based on CherryPy.

salt-api - A REST API for Salt, the infrastructure orchestration tool.

1.2.3 Products inspired by CherryPy

OOWeb - "OOWeb is a lightweight, embedded HTTP server for Java applications that maps objects to URL directories, methods to pages and form/querystring arguments as method parameters. OOWeb was originally inspired by CherryPy."

CHAPTER

TWO

INSTALLATION

CherryPy is a pure Python library. This has various consequences:

- It can run anywhere Python runs
- It does not require a C compiler
- It can run on various implementations of the Python language: CPython, IronPython, Jython and PyPy

Contents

2.1 Requirements

CherryPy does not have any mandatory env requirements. Python-based distribution requirements are installed automatically by `pip`. However certain features it comes with will require you install certain packages. To simplify installing additional dependencies CherryPy enables you to specify extras in your requirements (e.g. `cherrypy[json,routes_dispatcher,ssl]`):

- doc – for documentation related stuff
- json – for custom JSON processing library
- routes_dispatcher – routes for declarative URL mapping dispatcher
- ssl – for OpenSSL bindings, useful in Python environments not having the builtin `ssl` module
- testing
- memcached_session – enables memcached backend session
- xcgi

2.2 Supported python version

CherryPy supports Python 3.5 through to 3.8.

2.3 Installing

CherryPy can be easily installed via common Python package managers such as setuptools or pip.

```
$ easy_install cherrypy
```

```
$ pip install cherrypy
```

You may also get the latest CherryPy version by grabbing the source code from Github:

```
$ git clone https://github.com/cherrypy/cherrypy
$ cd cherrypy
$ python setup.py install
```

2.3.1 Test your installation

CherryPy comes with a set of simple tutorials that can be executed once you have deployed the package.

```
$ python -m cherrypy.tutorial.tut01_helloworld
```

Point your browser at http://127.0.0.1:8080 and enjoy the magic.

Once started the above command shows the following logs:

```
[15/Feb/2014:21:51:22] ENGINE Listening for SIGHUP.
[15/Feb/2014:21:51:22] ENGINE Listening for SIGTERM.
[15/Feb/2014:21:51:22] ENGINE Listening for SIGUSR1.
[15/Feb/2014:21:51:22] ENGINE Bus STARTING
[15/Feb/2014:21:51:22] ENGINE Started monitor thread 'Autoreloader'.
[15/Feb/2014:21:51:22] ENGINE Serving on http://127.0.0.1:8080
[15/Feb/2014:21:51:23] ENGINE Bus STARTED
```

We will explain what all those lines mean later on, but suffice to know that once you see the last two lines, your server is listening and ready to receive requests.

2.4 Run it

During development, the easiest path is to run your application as follow:

```
$ python myapp.py
```

As long as `myapp.py` defines a `"__main__"` section, it will run just fine.

2.4.1 cherryd

Another way to run the application is through the `cherryd` script which is installed along side CherryPy.

Note: This utility command will not concern you if you embed your application with another framework.

Command-Line Options

-c, --config
Specify config file(s)

-d
Run the server as a daemon

-e, --environment
Apply the given config environment (defaults to None)

-f
Start a *FastCGI* server instead of the default HTTP server

-s
Start a SCGI server instead of the default HTTP server

-i, --import
Specify modules to import

-p, --pidfile
Store the process id in the given file (defaults to None)

-P, --Path
Add the given paths to sys.path

CHAPTER

THREE

TUTORIALS

This tutorial will walk you through basic but complete CherryPy applications that will show you common concepts as well as slightly more advanced ones.

Contents

3.1 Tutorial 1: A basic web application

The following example demonstrates the most basic application you could write with CherryPy. It starts a server and hosts an application that will be served at request reaching http://127.0.0.1:8080/

```
import cherrypy


class HelloWorld(object):
    @cherrypy.expose
    def index(self):
        return "Hello world!"


if __name__ == '__main__':
    cherrypy.quickstart(HelloWorld())
```

Store this code snippet into a file named `tut01.py` and execute it as follows:

```
$ python tut01.py
```

This will display something along the following:

```
[24/Feb/2014:21:01:46] ENGINE Listening for SIGHUP.
[24/Feb/2014:21:01:46] ENGINE Listening for SIGTERM.
[24/Feb/2014:21:01:46] ENGINE Listening for SIGUSR1.
[24/Feb/2014:21:01:46] ENGINE Bus STARTING
CherryPy Checker:
The Application mounted at '' has an empty config.

[24/Feb/2014:21:01:46] ENGINE Started monitor thread 'Autoreloader'.
[24/Feb/2014:21:01:46] ENGINE Serving on http://127.0.0.1:8080
[24/Feb/2014:21:01:46] ENGINE Bus STARTED
```

This tells you several things. The first three lines indicate the server will handle `signal` for you. The next line tells you the current state of the server, as that point it is in `STARTING` stage. Then, you are notified your application has no specific configuration set to it. Next, the server starts a couple of internal utilities that we will explain later. Finally, the server indicates it is now ready to accept incoming communications as it listens on the address `127.0.0.1:8080`. In other words, at that stage your application is ready to be used.

Before moving on, let's discuss the message regarding the lack of configuration. By default, CherryPy has a feature which will review the syntax correctness of settings you could provide to configure the application. When none are provided, a warning message is thus displayed in the logs. That log is harmless and will not prevent CherryPy from working. You can refer to *the documentation above* to understand how to set the configuration.

3.2 Tutorial 2: Different URLs lead to different functions

Your applications will obviously handle more than a single URL. Let's imagine you have an application that generates a random string each time it is called:

```
import random
import string

import cherrypy
```

(continues on next page)

(continued from previous page)

```
5
6
7   class StringGenerator(object):
8       @cherrypy.expose
9       def index(self):
10          return "Hello world!"
11
12      @cherrypy.expose
13      def generate(self):
14          return ''.join(random.sample(string.hexdigits, 8))
15
16
17  if __name__ == '__main__':
18      cherrypy.quickstart(StringGenerator())
```

Save this into a file named `tut02.py` and run it as follows:

```
$ python tut02.py
```

Go now to http://localhost:8080/generate and your browser will display a random string.

Let's take a minute to decompose what's happening here. This is the URL that you have typed into your browser: http://localhost:8080/generate

This URL contains various parts:

- `http://` which roughly indicates it's a URL using the HTTP protocol (see **RFC 2616**).
- `localhost:8080` is the server's address. It's made of a hostname and a port.
- `/generate` which is the path segment of the URL. This is what CherryPy uses to locate an *exposed* function or method to respond.

Here CherryPy uses the *`index()`* method to handle `/` and the `generate()` method to handle `/generate`

3.3 Tutorial 3: My URLs have parameters

In the previous tutorial, we have seen how to create an application that could generate a random string. Let's now assume you wish to indicate the length of that string dynamically.

```
1   import random
2   import string
3
4   import cherrypy
5
6
7   class StringGenerator(object):
8       @cherrypy.expose
9       def index(self):
10          return "Hello world!"
11
12      @cherrypy.expose
13      def generate(self, length=8):
14          return ''.join(random.sample(string.hexdigits, int(length)))
15
16
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    cherrypy.quickstart(StringGenerator())
```

Save this into a file named `tut03.py` and run it as follows:

```
$ python tut03.py
```

Go now to http://localhost:8080/generate?length=16 and your browser will display a generated string of length 16. Notice how we benefit from Python's default arguments' values to support URLs such as http://localhost:8080/generate still.

In a URL such as this one, the section after `?` is called a query-string. Traditionally, the query-string is used to contextualize the URL by passing a set of (key, value) pairs. The format for those pairs is `key=value`. Each pair being separated by a `&` character.

Notice how we have to convert the given *`length`* value to an integer. Indeed, values are sent out from the client to our server as strings.

Much like CherryPy maps URL path segments to exposed functions, query-string keys are mapped to those exposed function parameters.

3.4 Tutorial 4: Submit this form

CherryPy is a web framework upon which you build web applications. The most traditional shape taken by applications is through an HTML user-interface speaking to your CherryPy server.

Let's see how to handle HTML forms via the following example.

```
import random
import string

import cherrypy


class StringGenerator(object):
    @cherrypy.expose
    def index(self):
        return """<html>
          <head></head>
          <body>
            <form method="get" action="generate">
              <input type="text" value="8" name="length" />
              <button type="submit">Give it now!</button>
            </form>
          </body>
        </html>"""

    @cherrypy.expose
    def generate(self, length=8):
        return ''.join(random.sample(string.hexdigits, int(length)))


if __name__ == '__main__':
    cherrypy.quickstart(StringGenerator())
```

Save this into a file named `tut04.py` and run it as follows:

```
$ python tut04.py
```

Go now to http://localhost:8080/ and your browser and this will display a simple input field to indicate the length of the string you want to generate.

Notice that in this example, the form uses the `GET` method and when you pressed the `Give it now!` button, the form is sent using the same URL as in the *previous* tutorial. HTML forms also support the `POST` method, in that case the query-string is not appended to the URL but it sent as the body of the client's request to the server. However, this would not change your application's exposed method because CherryPy handles both the same way and uses the exposed's handler parameters to deal with the query-string (key, value) pairs.

3.5 Tutorial 5: Track my end-user's activity

It's not uncommon that an application needs to follow the user's activity for a while. The usual mechanism is to use a session identifier that is carried during the conversation between the user and your application.

```
import random
import string

import cherrypy


class StringGenerator(object):
    @cherrypy.expose
    def index(self):
        return """<html>
          <head></head>
          <body>
            <form method="get" action="generate">
              <input type="text" value="8" name="length" />
              <button type="submit">Give it now!</button>
            </form>
          </body>
        </html>"""

    @cherrypy.expose
    def generate(self, length=8):
        some_string = ''.join(random.sample(string.hexdigits, int(length)))
        cherrypy.session['mystring'] = some_string
        return some_string

    @cherrypy.expose
    def display(self):
        return cherrypy.session['mystring']


if __name__ == '__main__':
    conf = {
        '/': {
            'tools.sessions.on': True
        }
    }
    cherrypy.quickstart(StringGenerator(), '/', conf)
```

Save this into a file named `tut05.py` and run it as follows:

```
$ python tut05.py
```

In this example, we generate the string as in the *previous* tutorial but also store it in the current session. If you go to http://localhost:8080/, generate a random string, then go to http://localhost:8080/display, you will see the string you just generated.

The lines 30-34 show you how to enable the session support in your CherryPy application. By default, CherryPy will save sessions in the process's memory. It supports more persistent *backends* as well.

3.6 Tutorial 6: What about my javascripts, CSS and images?

Web applications are usually also made of static content such as javascript, CSS files or images. CherryPy provides support to serve static content to end-users.

Let's assume, you want to associate a stylesheet with your application to display a blue background color (why not?).

First, save the following stylesheet into a file named `style.css` and stored into a local directory `public/css`.

```
1  body {
2    background-color: blue;
3  }
```

Now let's update the HTML code so that we link to the stylesheet using the http://localhost:8080/static/css/style.css URL.

```
1  import os, os.path
2  import random
3  import string
4
5  import cherrypy
6
7
8  class StringGenerator(object):
9      @cherrypy.expose
10     def index(self):
11         return """<html>
12           <head>
13             <link href="/static/css/style.css" rel="stylesheet">
14           </head>
15           <body>
16             <form method="get" action="generate">
17               <input type="text" value="8" name="length" />
18               <button type="submit">Give it now!</button>
19             </form>
20           </body>
21         </html>"""
22
23     @cherrypy.expose
24     def generate(self, length=8):
25         some_string = ''.join(random.sample(string.hexdigits, int(length)))
26         cherrypy.session['mystring'] = some_string
27         return some_string
28
29     @cherrypy.expose
30     def display(self):
31         return cherrypy.session['mystring']
```

(continues on next page)

(continued from previous page)

```
32
33
34  if __name__ == '__main__':
35      conf = {
36          '/': {
37              'tools.sessions.on': True,
38              'tools.staticdir.root': os.path.abspath(os.getcwd())
39          },
40          '/static': {
41              'tools.staticdir.on': True,
42              'tools.staticdir.dir': './public'
43          }
44      }
45      cherrypy.quickstart(StringGenerator(), '/', conf)
```

Save this into a file named `tut06.py` and run it as follows:

```
$ python tut06.py
```

Going to http://localhost:8080/, you should be greeted by a flashy blue color.

CherryPy provides support to serve a single file or a complete directory structure. Most of the time, this is what you'll end up doing so this is what the code above demonstrates. First, we indicate the *`root`* directory of all of our static content. This must be an absolute path for security reason. CherryPy will complain if you provide only relative paths when looking for a match to your URLs.

Then we indicate that all URLs which path segment starts with `/static` will be served as static content. We map that URL to the `public` directory, a direct child of the *`root`* directory. The entire sub-tree of the `public` directory will be served as static content. CherryPy will map URLs to path within that directory. This is why `/static/css/style.css` is found in `public/css/style.css`.

3.7 Tutorial 7: Give us a REST

It's not unusual nowadays that web applications expose some sort of datamodel or computation functions. Without going into its details, one strategy is to follow the REST principles edicted by Roy T. Fielding.

Roughly speaking, it assumes that you can identify a resource and that you can address that resource through that identifier.

"What for?" you may ask. Well, mostly, these principles are there to ensure that you decouple, as best as you can, the entities your application expose from the way they are manipulated or consumed. To embrace this point of view, developers will usually design a web API that expose pairs of `(URL, HTTP method, data, constraints)`.

Note: You will often hear REST and web API together. The former is one strategy to provide the latter. This tutorial will not go deeper in that whole web API concept as it's a much more engaging subject, but you ought to read more about it online.

Lets go through a small example of a very basic web API mildly following REST principles.

```
1  import random
2  import string
3
4  import cherrypy
```

(continues on next page)

(continued from previous page)

```
5
6
7   @cherrypy.expose
8   class StringGeneratorWebService(object):
9
10      @cherrypy.tools.accept(media='text/plain')
11      def GET(self):
12          return cherrypy.session['mystring']
13
14      def POST(self, length=8):
15          some_string = ''.join(random.sample(string.hexdigits, int(length)))
16          cherrypy.session['mystring'] = some_string
17          return some_string
18
19      def PUT(self, another_string):
20          cherrypy.session['mystring'] = another_string
21
22      def DELETE(self):
23          cherrypy.session.pop('mystring', None)
24
25
26  if __name__ == '__main__':
27      conf = {
28          '/': {
29              'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
30              'tools.sessions.on': True,
31              'tools.response_headers.on': True,
32              'tools.response_headers.headers': [('Content-Type', 'text/plain')],
33          }
34      }
35      cherrypy.quickstart(StringGeneratorWebService(), '/', conf)
```

Save this into a file named `tut07.py` and run it as follows:

```
$ python tut07.py
```

Before we see it in action, let's explain a few things. Until now, CherryPy was creating a tree of exposed methods that were used to match URLs. In the case of our web API, we want to stress the role played by the actual requests' HTTP methods. So we created methods that are named after them and they are all exposed at once by decorating the class itself with *`cherrypy.expose`*.

However, we must then switch from the default mechanism of matching URLs to method for one that is aware of the whole HTTP method shenanigan. This is what goes on line 27 where we create a `MethodDispatcher` instance.

Then we force the responses `content-type` to be `text/plain` and we finally ensure that `GET` requests will only be responded to clients that accept that `content-type` by having a `Accept: text/plain` header set in their request. However, we do this only for that HTTP method as it wouldn't have much meaning on the other methods.

For the purpose of this tutorial, we will be using a Python client rather than your browser as we wouldn't be able to actually try our web API otherwise.

Please install requests through the following command:

```
$ pip install requests
```

Then fire up a Python terminal and try the following commands:

```
>>> import requests
>>> s = requests.Session()
>>> r = s.get('http://127.0.0.1:8080/')
>>> r.status_code
500
>>> r = s.post('http://127.0.0.1:8080/')
>>> r.status_code, r.text
(200, u'04A92138')
>>> r = s.get('http://127.0.0.1:8080/')
>>> r.status_code, r.text
(200, u'04A92138')
>>> r = s.get('http://127.0.0.1:8080/', headers={'Accept': 'application/json'})
>>> r.status_code
406
>>> r = s.put('http://127.0.0.1:8080/', params={'another_string': 'hello'})
>>> r = s.get('http://127.0.0.1:8080/')
>>> r.status_code, r.text
(200, u'hello')
>>> r = s.delete('http://127.0.0.1:8080/')
>>> r = s.get('http://127.0.0.1:8080/')
>>> r.status_code
500
```

The first and last `500` responses stem from the fact that, in the first case, we haven't yet generated a string through `POST` and, on the latter case, that it doesn't exist after we've deleted it.

Lines 12-14 show you how the application reacted when our client requested the generated string as a JSON format. Since we configured the web API to only support plain text, it returns the appropriate HTTP error code.

Note: We use the Session interface of `requests` so that it takes care of carrying the session id stored in the request cookie in each subsequent request. That is handy.

Important: It's all about RESTful URLs these days, isn't it?

It is likely your URL will be made of dynamic parts that you will not be able to match to page handlers. For example, `/library/12/book/15` cannot be directly handled by the default CherryPy dispatcher since the segments `12` and `15` will not be matched to any Python callable.

This can be easily workaround with two handy CherryPy features explained in the *advanced section*.

3.8 Tutorial 8: Make it smoother with Ajax

In the recent years, web applications have moved away from the simple pattern of "HTML forms + refresh the whole page". This traditional scheme still works very well but users have become used to web applications that don't refresh the entire page. Broadly speaking, web applications carry code performed client-side that can speak with the backend without having to refresh the whole page.

This tutorial will involve a little more code this time around. First, let's see our CSS stylesheet located in `public/css/style.css`.

```
body {
  background-color: blue;
```

(continues on next page)

(continued from previous page)

```
}

#the-string {
  display: none;
}
```

We're adding a simple rule about the element that will display the generated string. By default, let's not show it up. Save the following HTML code into a file named `index.html`.

```
<!DOCTYPE html>
<html>
  <head>
    <link href="/static/css/style.css" rel="stylesheet">
    <script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>
    <script type="text/javascript">
      $(document).ready(function() {

        $("#generate-string").click(function(e) {
          $.post("/generator", {"length": $("input[name='length']").val()})
           .done(function(string) {
            $("#the-string").show();
            $("#the-string input").val(string);
          });
          e.preventDefault();
        });

        $("#replace-string").click(function(e) {
          $.ajax({
            type: "PUT",
            url: "/generator",
            data: {"another_string": $("#the-string input").val()}
          })
          .done(function() {
            alert("Replaced!");
          });
          e.preventDefault();
        });

        $("#delete-string").click(function(e) {
          $.ajax({
            type: "DELETE",
            url: "/generator"
          })
          .done(function() {
            $("#the-string").hide();
          });
          e.preventDefault();
        });

      });
    </script>
  </head>
  <body>
    <input type="text" value="8" name="length"/>
    <button id="generate-string">Give it now!</button>
    <div id="the-string">
```

(continues on next page)

(continued from previous page)

```
      <input type="text" />
      <button id="replace-string">Replace</button>
      <button id="delete-string">Delete it</button>
    </div>
  </body>
</html>
```

We'll be using the jQuery framework out of simplicity but feel free to replace it with your favourite tool. The page is composed of simple HTML elements to get user input and display the generated string. It also contains client-side code to talk to the backend API that actually performs the hard work.

Finally, here's the application's code that serves the HTML page above and responds to requests to generate strings. Both are hosted by the same application server.

```
import os, os.path
import random
import string

import cherrypy


class StringGenerator(object):
    @cherrypy.expose
    def index(self):
        return open('index.html')


@cherrypy.expose
class StringGeneratorWebService(object):

    @cherrypy.tools.accept(media='text/plain')
    def GET(self):
        return cherrypy.session['mystring']

    def POST(self, length=8):
        some_string = ''.join(random.sample(string.hexdigits, int(length)))
        cherrypy.session['mystring'] = some_string
        return some_string

    def PUT(self, another_string):
        cherrypy.session['mystring'] = another_string

    def DELETE(self):
        cherrypy.session.pop('mystring', None)


if __name__ == '__main__':
    conf = {
        '/': {
            'tools.sessions.on': True,
            'tools.staticdir.root': os.path.abspath(os.getcwd())
        },
        '/generator': {
            'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
            'tools.response_headers.on': True,
            'tools.response_headers.headers': [('Content-Type', 'text/plain')],
        },
```

(continues on next page)

(continued from previous page)

```
        '/static': {
            'tools.staticdir.on': True,
            'tools.staticdir.dir': './public'
        }
    }
    webapp = StringGenerator()
    webapp.generator = StringGeneratorWebService()
    cherrypy.quickstart(webapp, '/', conf)
```

Save this into a file named `tut08.py` and run it as follows:

```
$ python tut08.py
```

Go to http://127.0.0.1:8080/ and play with the input and buttons to generate, replace or delete the strings. Notice how the page isn't refreshed, simply part of its content.

Notice as well how your frontend converses with the backend using a straightfoward, yet clean, web service API. That same API could easily be used by non-HTML clients.

3.9 Tutorial 9: Data is all my life

Until now, all the generated strings were saved in the session, which by default is stored in the process memory. Though, you can persist sessions on disk or in a distributed memory store, this is not the right way of keeping your data on the long run. Sessions are there to identify your user and carry as little amount of data as necessary for the operation carried by the user.

To store, persist and query data you need a proper database server. There exist many to choose from with various paradigm support:

- relational: PostgreSQL, SQLite, MariaDB, Firebird
- column-oriented: HBase, Cassandra
- key-store: redis, memcached
- document oriented: Couchdb, MongoDB
- graph-oriented: neo4j

Let's focus on the relational ones since they are the most common and probably what you will want to learn first.

For the sake of reducing the number of dependencies for these tutorials, we will go for the `sqlite` database which is directly supported by Python.

Our application will replace the storage of the generated string from the session to a SQLite database. The application will have the same HTML code as *tutorial 08*. So let's simply focus on the application code itself:

```
import os, os.path
import random
import sqlite3
import string
import time

import cherrypy

DB_STRING = "my.db"

```

(continues on next page)

(continued from previous page)

```

class StringGenerator(object):
    @cherrypy.expose
    def index(self):
        return open('index.html')


@cherrypy.expose
class StringGeneratorWebService(object):

    @cherrypy.tools.accept(media='text/plain')
    def GET(self):
        with sqlite3.connect(DB_STRING) as c:
            cherrypy.session['ts'] = time.time()
            r = c.execute("SELECT value FROM user_string WHERE session_id=?",
                          [cherrypy.session.id])
            return r.fetchone()

    def POST(self, length=8):
        some_string = ''.join(random.sample(string.hexdigits, int(length)))
        with sqlite3.connect(DB_STRING) as c:
            cherrypy.session['ts'] = time.time()
            c.execute("INSERT INTO user_string VALUES (?, ?)",
                      [cherrypy.session.id, some_string])
        return some_string

    def PUT(self, another_string):
        with sqlite3.connect(DB_STRING) as c:
            cherrypy.session['ts'] = time.time()
            c.execute("UPDATE user_string SET value=? WHERE session_id=?",
                      [another_string, cherrypy.session.id])

    def DELETE(self):
        cherrypy.session.pop('ts', None)
        with sqlite3.connect(DB_STRING) as c:
            c.execute("DELETE FROM user_string WHERE session_id=?",
                      [cherrypy.session.id])


def setup_database():
    """
    Create the `user_string` table in the database
    on server startup
    """
    with sqlite3.connect(DB_STRING) as con:
        con.execute("CREATE TABLE user_string (session_id, value)")


def cleanup_database():
    """
    Destroy the `user_string` table from the database
    on server shutdown.
    """
    with sqlite3.connect(DB_STRING) as con:
        con.execute("DROP TABLE user_string")


```

(continues on next page)

(continued from previous page)

```
68 if __name__ == '__main__':
69     conf = {
70         '/': {
71             'tools.sessions.on': True,
72             'tools.staticdir.root': os.path.abspath(os.getcwd())
73         },
74         '/generator': {
75             'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
76             'tools.response_headers.on': True,
77             'tools.response_headers.headers': [('Content-Type', 'text/plain')],
78         },
79         '/static': {
80             'tools.staticdir.on': True,
81             'tools.staticdir.dir': './public'
82         }
83     }
84
85     cherrypy.engine.subscribe('start', setup_database)
86     cherrypy.engine.subscribe('stop', cleanup_database)
87
88     webapp = StringGenerator()
89     webapp.generator = StringGeneratorWebService()
90     cherrypy.quickstart(webapp, '/', conf)
```

Save this into a file named `tut09.py` and run it as follows:

```
$ python tut09.py
```

Let's first see how we create two functions that create and destroy the table within our database. These functions are registered to the CherryPy's server on lines 85-86, so that they are called when the server starts and stops.

Next, notice how we replaced all the session code with calls to the database. We use the session id to identify the user's string within our database. Since the session will go away after a while, it's probably not the right approach. A better idea would be to associate the user's login or more resilient unique identifier. For the sake of our demo, this should do.

Important: In this example, we must still set the session to a dummy value so that the session is not discarded on each request by CherryPy. Since we now use the database to store the generated string, we simply store a dummy timestamp inside the session.

Note: Unfortunately, sqlite in Python forbids us to share a connection between threads. Since CherryPy is a multi-threaded server, this would be an issue. This is the reason why we open and close a connection to the database on each call. This is clearly not really production friendly, and it is probably advisable to either use a more capable database engine or a higher level library, such as SQLAlchemy, to better support your application's needs.

3.10 Tutorial 10: Make it a modern single-page application with React.js

In the recent years, client-side single-page applications (SPA) have gradually eaten server-side generated content web applications's lunch.

This tutorial demonstrates how to integrate with React.js, a Javascript library for SPA released by Facebook in 2013. Please refer to React.js documentation to learn more about it.

To demonstrate it, let's use the code from *tutorial 09*. However, we will be replacing the HTML and Javascript code.

First, let's see how our HTML code has changed:

```
<!DOCTYPE html>
<html>
    <head>
      <link href="/static/css/style.css" rel="stylesheet">
      <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/react.js"></
↪script>
      <script src="http://code.jquery.com/jquery-2.1.1.min.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.
↪min.js"></script>
    </head>
    <body>
      <div id="generator"></div>
      <script type="text/babel" src="static/js/gen.js"></script>
    </body>
</html>
```

Basically, we have removed the entire Javascript code that was using jQuery. Instead, we load the React.js library as well as a new, local, Javascript module, named `gen.js` and located in the `public/js` directory:

```
var StringGeneratorBox = React.createClass({
  handleGenerate: function() {
    var length = this.state.length;
    this.setState(function() {
      $.ajax({
        url: this.props.url,
        dataType: 'text',
        type: 'POST',
        data: {
          "length": length
        },
        success: function(data) {
          this.setState({
            length: length,
            string: data,
            mode: "edit"
          });
        }.bind(this),
        error: function(xhr, status, err) {
          console.error(this.props.url,
            status, err.toString()
          );
        }.bind(this)
      });
    });
```

(continues on next page)

(continued from previous page)

```
  },
  handleEdit: function() {
    var new_string = this.state.string;
    this.setState(function() {
      $.ajax({
        url: this.props.url,
        type: 'PUT',
        data: {
          "another_string": new_string
        },
        success: function() {
          this.setState({
            length: new_string.length,
            string: new_string,
            mode: "edit"
          });
        }.bind(this),
        error: function(xhr, status, err) {
          console.error(this.props.url,
            status, err.toString()
          );
        }.bind(this)
      });
    });
  },
  handleDelete: function() {
    this.setState(function() {
      $.ajax({
        url: this.props.url,
        type: 'DELETE',
        success: function() {
          this.setState({
            length: "8",
            string: "",
            mode: "create"
          });
        }.bind(this),
        error: function(xhr, status, err) {
          console.error(this.props.url,
            status, err.toString()
          );
        }.bind(this)
      });
    });
  },
  handleLengthChange: function(length) {
    this.setState({
      length: length,
      string: "",
      mode: "create"
    });
  },
  handleStringChange: function(new_string) {
    this.setState({
      length: new_string.length,
      string: new_string,
      mode: "edit"
```

(continues on next page)

(continued from previous page)

```
    });
  },
  getInitialState: function() {
    return {
      length: "8",
      string: "",
      mode: "create"
    };
  },
  render: function() {
    return (
      <div className="stringGenBox">
            <StringGeneratorForm onCreateString={this.handleGenerate}
                                 onReplaceString={this.handleEdit}
                                 onDeleteString={this.handleDelete}
                                 onLengthChange={this.handleLengthChange}
                                 onStringChange={this.handleStringChange}
                                 mode={this.state.mode}
                                 length={this.state.length}
                                 string={this.state.string}/>
      </div>
    );
  }
});

var StringGeneratorForm = React.createClass({
  handleCreate: function(e) {
    e.preventDefault();
    this.props.onCreateString();
  },
  handleReplace: function(e) {
    e.preventDefault();
    this.props.onReplaceString();
  },
  handleDelete: function(e) {
    e.preventDefault();
    this.props.onDeleteString();
  },
  handleLengthChange: function(e) {
    e.preventDefault();
    var length = React.findDOMNode(this.refs.length).value.trim();
    this.props.onLengthChange(length);
  },
  handleStringChange: function(e) {
    e.preventDefault();
    var string = React.findDOMNode(this.refs.string).value.trim();
    this.props.onStringChange(string);
  },
  render: function() {
    if (this.props.mode == "create") {
      return (
        <div>
           <input  type="text" ref="length" defaultValue="8" value={this.props.length}
↪ onChange={this.handleLengthChange} />
           <button onClick={this.handleCreate}>Give it now!</button>
        </div>
      );
```

(continues on next page)

(continued from previous page)

```
    } else if (this.props.mode == "edit") {
      return (
        <div>
           <input type="text" ref="string" value={this.props.string} onChange={this.
↪handleStringChange} />
           <button onClick={this.handleReplace}>Replace</button>
           <button onClick={this.handleDelete}>Delete it</button>
        </div>
      );
    }

    return null;
  }
});

React.render(
  <StringGeneratorBox url="/generator" />,
  document.getElementById('generator')
);
```

Wow! What a lot of code for something so simple, isn't it? The entry point is the last few lines where we indicate that we want to render the HTML code of the `StringGeneratorBox` React.js class inside the `generator` div.

When the page is rendered, so is that component. Notice how it is also made of another component that renders the form itself.

This might be a little over the top for such a simple example but hopefully will get you started with React.js in the process.

There is not much to say and, hopefully, the meaning of that code is rather clear. The component has an internal state in which we store the current string as generated/modified by the user.

When the user changes the content of the input boxes, the state is updated on the client side. Then, when a button is clicked, that state is sent out to the backend server using the API endpoint and the appropriate action takes places. Then, the state is updated and so is the view.

3.11 Tutorial 11: Organize my code

CherryPy comes with a powerful architecture that helps you organizing your code in a way that should make it easier to maintain and more flexible.

Several mechanisms are at your disposal, this tutorial will focus on the three main ones:

- *dispatchers*
- *tools*
- *plugins*

In order to understand them, let's imagine you are at a superstore:

- You have several tills and people queuing for each of them (those are your requests)
- You have various sections with food and other stuff (these are your data)
- Finally you have the superstore people and their daily tasks to make sure sections are always in order (this is your backend)

In spite of being really simplistic, this is not far from how your application behaves. CherryPy helps you structure your application in a way that mirrors these high-level ideas.

3.11.1 Dispatchers

Coming back to the superstore example, it is likely that you will want to perform operations based on the till:

- Have a till for baskets with less than ten items
- Have a till for disabled people
- Have a till for pregnant women
- Have a till where you can only using the store card

To support these use-cases, CherryPy provides a mechanism called a *dispatcher*. A dispatcher is executed early during the request processing in order to determine which piece of code of your application will handle the incoming request. Or, to continue on the store analogy, a dispatcher will decide which till to lead a customer to.

3.11.2 Tools

Let's assume your store has decided to operate a discount spree but, only for a specific category of customers. CherryPy will deal with such use case via a mechanism called a *tool*.

A tool is a piece of code that runs on a per-request basis in order to perform additional work. Usually a tool is a simple Python function that is executed at a given point during the process of the request by CherryPy.

3.11.3 Plugins

As we have seen, the store has a crew of people dedicated to manage the stock and deal with any customers' expectation.

In the CherryPy world, this translates into having functions that run outside of any request life-cycle. These functions should take care of background tasks, long lived connections (such as those to a database for instance), etc.

Plugins are called that way because they work along with the CherryPy *engine* and extend it with your operations.

3.12 Tutorial 12: Using pytest and code coverage

3.12.1 Pytest

Let's revisit *Tutorial 2*.

```
1  import random
2  import string
3
4  import cherrypy
5
6
7  class StringGenerator(object):
8      @cherrypy.expose
9      def index(self):
10         return "Hello world!"
11
```

(continues on next page)

(continued from previous page)

```
    @cherrypy.expose
    def generate(self):
        return ''.join(random.sample(string.hexdigits, 8))


if __name__ == '__main__':
    cherrypy.quickstart(StringGenerator())
```

Save this into a file named `tut12.py`.

Now make the test file:

```
import cherrypy
from cherrypy.test import helper

from tut12 import StringGenerator

class SimpleCPTest(helper.CPWebCase):
    @staticmethod
    def setup_server():
        cherrypy.tree.mount(StringGenerator(), '/', {})

    def test_index(self):
        self.getPage("/")
        self.assertStatus('200 OK')
    def test_generate(self):
        self.getPage("/generate")
        self.assertStatus('200 OK')
```

Save this into a file named `test_tut12.py` and run

```
$ pytest -v test_tut12.py
```

Note: If you don't have pytest installed, you'll need to install it by `pip install pytest`

We now have a neat way that we can exercise our application making tests.

3.12.2 Adding Code Coverage

To get code coverage, simply run

```
$ pytest --cov=tut12 --cov-report term-missing test_tut12.py
```

Note: To add coverage support to pytest, you'll need to install it by `pip install pytest-cov`

This tells us that one line is missing. Of course it is because that is only executed when the python program is started directly. We can simply change the following lines in `tut12.py`:

```
if __name__ == '__main__':  # pragma: no cover
    cherrypy.quickstart(StringGenerator())
```

When you rerun the code coverage, it should show 100% now.

Note: When using in CI, you might want to integrate Codecov, Landscape or Coveralls into your project to store and track coverage data over time.

CHAPTER

FOUR

BASICS

The following sections will drive you through the basics of a CherryPy application, introducing some essential concepts.

Contents

4.1 The one-minute application example

The most basic application you can write with CherryPy involves almost all its core concepts.

```
1  import cherrypy
2
3  class Root(object):
4      @cherrypy.expose
5      def index(self):
6          return "Hello World!"
7
8  if __name__ == '__main__':
9      cherrypy.quickstart(Root(), '/')
```

First and foremost, for most tasks, you will never need more than a single import statement as demonstrated in line 1.

Before discussing the meat, let's jump to line 9 which shows, how to host your application with the CherryPy application server and serve it with its builtin HTTP server at the `'/'` path. All in one single line. Not bad.

Let's now step back to the actual application. Even though CherryPy does not mandate it, most of the time your applications will be written as Python classes. Methods of those classes will be called by CherryPy to respond to client requests. However, CherryPy needs to be aware that a method can be used that way, we say the method needs to be *exposed*. This is precisely what the *`cherrypy.expose()`* decorator does in line 4.

Save the snippet in a file named `myapp.py` and run your first CherryPy application:

```
$ python myapp.py
```

Then point your browser at http://127.0.0.1:8080. Tada!

Note: CherryPy is a small framework that focuses on one single task: take a HTTP request and locate the most appropriate Python function or method that match the request's URL. Unlike other well-known frameworks, CherryPy does not provide a built-in support for database access, HTML templating or any other middleware nifty features.

In a nutshell, once CherryPy has found and called an *exposed* method, it is up to you, as a developer, to provide the tools to implement your application's logic.

CherryPy takes the opinion that you, the developer, know best.

Warning: The previous example demonstrated the simplicty of the CherryPy interface but, your application will likely contain a few other bits and pieces: static service, more complex structure, database access, etc. This will be developed in the tutorial section.

CherryPy is a minimal framework but not a bare one, it comes with a few basic tools to cover common usages that you would expect.

4.2 Hosting one or more applications

A web application needs an HTTP server to be accessed to. CherryPy provides its own, production ready, HTTP server. There are two ways to host an application with it. The simple one and the almost-as-simple one.

4.2.1 Single application

The most straightforward way is to use *cherrypy.quickstart()* function. It takes at least one argument, the instance of the application to host. Two other settings are optionals. First, the base path at which the application will be accessible from. Second, a config dictionary or file to configure your application.

```
cherrypy.quickstart(Blog())
cherrypy.quickstart(Blog(), '/blog')
cherrypy.quickstart(Blog(), '/blog', {'/': {'tools.gzip.on': True}})
```

The first one means that your application will be available at http://hostname:port/ whereas the other two will make your blog application available at http://hostname:port/blog. In addition, the last one provides specific settings for the application.

Note: Notice in the third case how the settings are still relative to the application, not where it is made available at, hence the `{'/': ... }` rather than a `{'/blog': ... }`

4.2.2 Multiple applications

The *cherrypy.quickstart()* approach is fine for a single application, but lacks the capacity to host several applications with the server. To achieve this, one must use the *cherrypy.tree.mount* function as follows:

```
cherrypy.tree.mount(Blog(), '/blog', blog_conf)
cherrypy.tree.mount(Forum(), '/forum', forum_conf)

cherrypy.engine.start()
cherrypy.engine.block()
```

Essentially, *cherrypy.tree.mount* takes the same parameters as *cherrypy.quickstart()*: an *application*, a hosting path segment and a configuration. The last two lines are simply starting application server.

Important: *cherrypy.quickstart()* and *cherrypy.tree.mount* are not exclusive. For instance, the previous lines can be written as:

```
cherrypy.tree.mount(Blog(), '/blog', blog_conf)
cherrypy.quickstart(Forum(), '/forum', forum_conf)
```

Note: You can also *host foreign WSGI application*.

4.3 Logging

Logging is an important task in any application. CherryPy will log all incoming requests as well as protocol errors.

To do so, CherryPy manages two loggers:

- an access one that logs every incoming requests
- an application/error log that traces errors or other application-level messages

Your application may leverage that second logger by calling *cherrypy.log()*.

```
cherrypy.log("hello there")
```

You can also log an exception:

```
try:
   ...
except Exception:
   cherrypy.log("kaboom!", traceback=True)
```

Both logs are writing to files identified by the following keys in your configuration:

- `log.access_file` for incoming requests using the common log format
- `log.error_file` for the other log

See also:

Refer to the *cherrypy._cplogging* module for more details about CherryPy's logging architecture.

4.3.1 Disable logging

You may be interested in disabling either logs.

To disable file logging, simply set a en empty string to the `log.access_file` or `log.error_file` keys in your *global configuration*.

To disable, console logging, set `log.screen` to `False`.

```
cherrypy.config.update({'log.screen': False,
                        'log.access_file': '',
                        'log.error_file': ''})
```

4.3.2 Play along with your other loggers

Your application may obviously already use the `logging` module to trace application level messages. Below is a simple example on setting it up.

```
import logging
import logging.config

import cherrypy

logger = logging.getLogger()
db_logger = logging.getLogger('db')

LOG_CONF = {
    'version': 1,

    'formatters': {
        'void': {
            'format': ''
        },
        'standard': {
            'format': '%(asctime)s [%(levelname)s] %(name)s: %(message)s'
        },
    },
    'handlers': {
        'default': {
            'level':'INFO',
            'class':'logging.StreamHandler',
            'formatter': 'standard',
            'stream': 'ext://sys.stdout'
        },
        'cherrypy_console': {
            'level':'INFO',
            'class':'logging.StreamHandler',
            'formatter': 'void',
            'stream': 'ext://sys.stdout'
        },
        'cherrypy_access': {
            'level':'INFO',
            'class': 'logging.handlers.RotatingFileHandler',
            'formatter': 'void',
            'filename': 'access.log',
            'maxBytes': 10485760,
            'backupCount': 20,
            'encoding': 'utf8'
        },
        'cherrypy_error': {
            'level':'INFO',
            'class': 'logging.handlers.RotatingFileHandler',
            'formatter': 'void',
            'filename': 'errors.log',
            'maxBytes': 10485760,
            'backupCount': 20,
            'encoding': 'utf8'
        },
    },
    'loggers': {
        '': {
```

(continues on next page)

(continued from previous page)

```
            'handlers': ['default'],
            'level': 'INFO'
        },
        'db': {
            'handlers': ['default'],
            'level': 'INFO' ,
            'propagate': False
        },
        'cherrypy.access': {
            'handlers': ['cherrypy_access'],
            'level': 'INFO',
            'propagate': False
        },
        'cherrypy.error': {
            'handlers': ['cherrypy_console', 'cherrypy_error'],
            'level': 'INFO',
            'propagate': False
        },
    }
}

class Root(object):
    @cherrypy.expose
    def index(self):

        logger.info("boom")
        db_logger.info("bam")
        cherrypy.log("bang")

        return "hello world"

if __name__ == '__main__':
    cherrypy.config.update({'log.screen': False,
                            'log.access_file': '',
                            'log.error_file': ''})
cherrypy.engine.unsubscribe('graceful', cherrypy.log.reopen_files)
    logging.config.dictConfig(LOG_CONF)
    cherrypy.quickstart(Root())
```

In this snippet, we create a configuration dictionary that we pass on to the `logging` module to configure our loggers:

- the default root logger is associated to a single stream handler
- a logger for the db backend with also a single stream handler

In addition, we re-configure the CherryPy loggers:

- the top-level `cherrypy.access` logger to log requests into a file
- the `cherrypy.error` logger to log everything else into a file and to the console

We also prevent CherryPy from trying to open its log files when the autoreloader kicks in. This is not strictly required since we do not even let CherryPy open them in the first place. But, this avoids wasting time on something useless.

4.4 Configuring

CherryPy comes with a fine-grained configuration mechanism and settings can be set at various levels.

See also:

Once you have the reviewed the basics, please refer to the *in-depth discussion* around configuration.

4.4.1 Global server configuration

To configure the HTTP and application servers, use the *cherrypy.config.update()* method.

```
cherrypy.config.update({'server.socket_port': 9090})
```

The *cherrypy.config* object is a dictionary and the update method merges the passed dictionary into it.

You can also pass a file instead (assuming a `server.conf` file):

```
[global]
server.socket_port: 9090
```

```
cherrypy.config.update("server.conf")
```

> **Warning:** *cherrypy.config.update()* is not meant to be used to configure the application. It is a common mistake. It is used to configure the server and engine.

4.4.2 Per-application configuration

To configure your application, pass in a dictionary or a file when you associate your application to the server.

```
cherrypy.quickstart(myapp, '/', {'/': {'tools.gzip.on': True}})
```

or via a file (called `app.conf` for instance):

```
[/]
tools.gzip.on: True
```

```
cherrypy.quickstart(myapp, '/', "app.conf")
```

Although, you can define most of your configuration in a global fashion, it is sometimes convenient to define them where they are applied in the code.

```
class Root(object):
    @cherrypy.expose
    @cherrypy.tools.gzip()
    def index(self):
        return "hello world!"
```

A variant notation to the above:

```
class Root(object):
    @cherrypy.expose
    def index(self):
        return "hello world!"
    index._cp_config = {'tools.gzip.on': True}
```

Both methods have the same effect so pick the one that suits your style best.

4.4.3 Additional application settings

You can add settings that are not specific to a request URL and retrieve them from your page handler as follows:

```
[/]
tools.gzip.on: True

[googleapi]
key = "..."
appid = "..."
```

```
class Root(object):
    @cherrypy.expose
    def index(self):
        google_appid = cherrypy.request.app.config['googleapi']['appid']
        return "hello world!"

cherrypy.quickstart(Root(), '/', "app.conf")
```

4.5 Cookies

CherryPy uses the `Cookie` module from python and in particular the `Cookie.SimpleCookie` object type to handle cookies.

- To send a cookie to a browser, set `cherrypy.response.cookie[key] = value`.
- To retrieve a cookie sent by a browser, use `cherrypy.request.cookie[key]`.
- To delete a cookie (on the client side), you must *send* the cookie with its expiration time set to `0`:

```
cherrypy.response.cookie[key] = value
cherrypy.response.cookie[key]['expires'] = 0
```

It's important to understand that the request cookies are **not** automatically copied to the response cookies. Clients will send the same cookies on every request, and therefore `cherrypy.request.cookie` should be populated each time. But the server doesn't need to send the same cookies with every response; therefore, `cherrypy.response.cookie` will usually be empty. When you wish to "delete" (expire) a cookie, therefore, you must set `cherrypy.response.cookie[key] = value` first, and then set its `expires` attribute to 0.

Extended example:

```
import cherrypy

class MyCookieApp(object):
    @cherrypy.expose
    def set(self):
```

(continues on next page)

(continued from previous page)

```
        cookie = cherrypy.response.cookie
        cookie['cookieName'] = 'cookieValue'
        cookie['cookieName']['path'] = '/'
        cookie['cookieName']['max-age'] = 3600
        cookie['cookieName']['version'] = 1
        return "<html><body>Hello, I just sent you a cookie</body></html>"

    @cherrypy.expose
    def read(self):
        cookie = cherrypy.request.cookie
        res = """<html><body>Hi, you sent me %s cookies.<br />
                Here is a list of cookie names/values:<br />""" % len(cookie)
        for name in cookie.keys():
            res += "name: %s, value: %s<br>" % (name, cookie[name].value)
        return res + "</body></html>"

if __name__ == '__main__':
    cherrypy.quickstart(MyCookieApp(), '/cookie')
```

4.6 Using sessions

Sessions are one of the most common mechanism used by developers to identify users and synchronize their activity. By default, CherryPy does not activate sessions because it is not a mandatory feature to have, to enable it simply add the following settings in your configuration:

```
[/]
tools.sessions.on: True
```

```
cherrypy.quickstart(myapp, '/', "app.conf")
```

Sessions are, by default, stored in RAM so, if you restart your server all of your current sessions will be lost. You can store them in memcached or on the filesystem instead.

Using sessions in your applications is done as follows:

```
import cherrypy

@cherrypy.expose
def index(self):
    if 'count' not in cherrypy.session:
       cherrypy.session['count'] = 0
    cherrypy.session['count'] += 1
```

In this snippet, everytime the index page handler is called, the current user's session has its `'count'` key incremented by `1`.

CherryPy knows which session to use by inspecting the cookie sent alongside the request. This cookie contains the session identifier used by CherryPy to load the user's session from the storage.

See also:

Refer to the *cherrypy.lib.sessions* module for more details about the session interface and implementation. Notably you will learn about sessions expiration.

4.6.1 Filesystem backend

Using a filesystem is a simple to not lose your sessions between reboots. Each session is saved in its own file within the given directory.

```
[/]
tools.sessions.on: True
tools.sessions.storage_class = cherrypy.lib.sessions.FileSession
tools.sessions.storage_path = "/some/directory"
```

4.6.2 Memcached backend

Memcached is a popular key-store on top of your RAM, it is distributed and a good choice if you want to share sessions outside of the process running CherryPy.

Requires that the Python memcached package is installed, which may be indicated by installing `cherrypy[memcached_session]`.

```
[/]
tools.sessions.on: True
tools.sessions.storage_class = cherrypy.lib.sessions.MemcachedSession
```

4.6.3 Other backends

Any other library may implement a session backend. Simply subclass `cherrypy.lib.sessions.Session` and indicate that subclass as `tools.sessions.storage_class`.

4.7 Static content serving

CherryPy can serve your static content such as images, javascript and CSS resources, etc.

Note: CherryPy uses the `mimetypes` module to determine the best content-type to serve a particular resource. If the choice is not valid, you can simply set more media-types as follows:

```
import mimetypes
mimetypes.types_map['.csv'] = 'text/csv'
```

4.7.1 Serving a single file

You can serve a single file as follows:

```
[/style.css]
tools.staticfile.on = True
tools.staticfile.filename = "/home/site/style.css"
```

CherryPy will automatically respond to URLs such as `http://hostname/style.css`.

4.7.2 Serving a whole directory

Serving a whole directory is similar to a single file:

```
[/static]
tools.staticdir.on = True
tools.staticdir.dir = "/home/site/static"
```

Assuming you have a file at `static/js/my.js`, CherryPy will automatically respond to URLs such as `http://hostname/static/js/my.js`.

Note: CherryPy always requires the absolute path to the files or directories it will serve. If you have several static sections to configure but located in the same root directory, you can use the following shortcut:

```
[/]
tools.staticdir.root = "/home/site"

[/static]
tools.staticdir.on = True
tools.staticdir.dir = "static"
```

4.7.3 Specifying an index file

By default, CherryPy will respond to the root of a static directory with an 404 error indicating the path '/' was not found. To specify an index file, you can use the following:

```
[/static]
tools.staticdir.on = True
tools.staticdir.dir = "/home/site/static"
tools.staticdir.index = "index.html"
```

Assuming you have a file at `static/index.html`, CherryPy will automatically respond to URLs such as `http://hostname/static/` by returning its contents.

4.7.4 Allow files downloading

Using `"application/x-download"` response content-type, you can tell a browser that a resource should be downloaded onto the user's machine rather than displayed.

You could for instance write a page handler as follows:

```
from cherrypy.lib.static import serve_file

@cherrypy.expose
def download(self, filepath):
    return serve_file(filepath, "application/x-download", "attachment")
```

Assuming the filepath is a valid path on your machine, the response would be considered as a downloadable content by the browser.

Warning: The above page handler is a security risk on its own since any file of the server could be accessed (if the user running the server had permissions on them).

4.8 Dealing with JSON

CherryPy has built-in support for JSON encoding and decoding of the request and/or response.

4.8.1 Decoding request

To automatically decode the content of a request using JSON:

```
class Root(object):
    @cherrypy.expose
    @cherrypy.tools.json_in()
    def index(self):
        data = cherrypy.request.json
```

The `json` attribute attached to the request contains the decoded content.

4.8.2 Encoding response

To automatically encode the content of a response using JSON:

```
class Root(object):
    @cherrypy.expose
    @cherrypy.tools.json_out()
    def index(self):
        return {'key': 'value'}
```

CherryPy will encode any content returned by your page handler using JSON. Not all type of objects may natively be encoded.

4.9 Authentication

CherryPy provides support for two very simple HTTP-based authentication mechanisms, described in **RFC 7616** and **RFC 7617** (which obsoletes **RFC 2617**): Basic and Digest. They are most commonly known to trigger a browser's popup asking users their name and password.

4.9.1 Basic

Basic authentication is the simplest form of authentication however it is not a secure one as the user's credentials are embedded into the request. We advise against using it unless you are running on SSL or within a closed network.

```
from cherrypy.lib import auth_basic

USERS = {'jon': 'secret'}

def validate_password(realm, username, password):
    if username in USERS and USERS[username] == password:
       return True
    return False

conf = {
   '/protected/area': {
```

(continues on next page)

(continued from previous page)

```
        'tools.auth_basic.on': True,
        'tools.auth_basic.realm': 'localhost',
        'tools.auth_basic.checkpassword': validate_password,
        'tools.auth_basic.accept_charset': 'UTF-8',
    }
}

cherrypy.quickstart(myapp, '/', conf)
```

Simply put, you have to provide a function that will be called by CherryPy passing the username and password decoded from the request.

The function can read its data from any source it has to: a file, a database, memory, etc.

4.9.2 Digest

Digest authentication differs by the fact the credentials are not carried on by the request so it's a little more secure than basic.

CherryPy's digest support has a similar interface to the basic one explained above.

```
from cherrypy.lib import auth_digest

USERS = {'jon': 'secret'}

conf = {
   '/protected/area': {
         'tools.auth_digest.on': True,
         'tools.auth_digest.realm': 'localhost',
         'tools.auth_digest.get_ha1': auth_digest.get_ha1_dict_plain(USERS),
         'tools.auth_digest.key': 'a565c27146791cfb',
         'tools.auth_digest.accept_charset': 'UTF-8',
   }
}

cherrypy.quickstart(myapp, '/', conf)
```

4.9.3 SO_PEERCRED

There's also a low-level authentication for UNIX file and abstract sockets. This is how you enable it:

```
[global]
server.peercreds: True
server.peercreds_resolve: True
server.socket_file: /var/run/cherrypy.sock
```

`server.peercreds` enables looking up the connected process ID, user ID and group ID. They'll be accessible as WSGI environment variables:

- `X_REMOTE_PID`
- `X_REMOTE_UID`
- `X_REMOTE_GID`

`server.peercreds_resolve` resolves that into user name and group name. They'll be accessible as WSGI environment variables:

- `X_REMOTE_USER` and `REMOTE_USER`
- `X_REMOTE_GROUP`

4.10 Favicon

CherryPy serves its own sweet red cherrypy as the default favicon using the static file tool. You can serve your own favicon as follows:

```
import cherrypy

class HelloWorld(object):
   @cherrypy.expose
   def index(self):
       return "Hello World!"

if __name__ == '__main__':
    cherrypy.quickstart(HelloWorld(), '/',
        {
            '/favicon.ico':
            {
                'tools.staticfile.on': True,
                'tools.staticfile.filename': '/path/to/myfavicon.ico'
            }
        }
    )
```

Please refer to the *static serving* section for more details.

You can also use a file to configure it:

```
[/favicon.ico]
tools.staticfile.on: True
tools.staticfile.filename: "/path/to/myfavicon.ico"
```

```
import cherrypy

class HelloWorld(object):
   @cherrypy.expose
   def index(self):
       return "Hello World!"

if __name__ == '__main__':
    cherrypy.quickstart(HelloWorld(), '/', "app.conf")
```

CHAPTER

FIVE

ADVANCED

CherryPy has support for more advanced features that these sections will describe.

Contents

5.1 Set aliases to page handlers

A fairly unknown, yet useful, feature provided by the *cherrypy.expose()* decorator is to support aliases.

Let's use the template provided by *tutorial 03*:

```
import random
import string

import cherrypy

class StringGenerator(object):
    @cherrypy.expose(['generer', 'generar'])
    def generate(self, length=8):
        return ''.join(random.sample(string.hexdigits, int(length)))

if __name__ == '__main__':
    cherrypy.quickstart(StringGenerator())
```

In this example, we create localized aliases for the page handler. This means the page handler will be accessible via:

- /generate
- /generer (French)
- /generar (Spanish)

Obviously, your aliases may be whatever suits your needs.

Note: The alias may be a single string or a list of them.

5.2 RESTful-style dispatching

The term `RESTful URL` is sometimes used to talk about friendly URLs that nicely map to the entities an application exposes.

Important: We will not enter the debate around what is restful or not but we will showcase two mechanisms to implement the usual idea in your CherryPy application.

Let's assume you wish to create an application that exposes music bands and their records. Your application will probably have the following URLs:

- http://hostname/<artist>/
- http://hostname/<artist>/albums/<album_title>/

It's quite clear you would not create a page handler named after every possible band in the world. This means you will need a page handler that acts as a proxy for all of them.

The default dispatcher cannot deal with that scenario on its own because it expects page handlers to be explicitly declared in your source code. Luckily, CherryPy provides ways to support those use cases.

See also:

This section extends from this stackoverflow response.

5.2.1 The special _cp_dispatch method

`_cp_dispatch` is a special method you declare in any of your *controller* to massage the remaining segments before CherryPy gets to process them. This offers you the capacity to remove, add or otherwise handle any segment you wish and, even, entirely change the remaining parts.

```
import cherrypy

class Band(object):
    def __init__(self):
        self.albums = Album()

    def _cp_dispatch(self, vpath):
        if len(vpath) == 1:
            cherrypy.request.params['name'] = vpath.pop()
            return self

        if len(vpath) == 3:
            cherrypy.request.params['artist'] = vpath.pop(0)  # /band name/
            vpath.pop(0) # /albums/
            cherrypy.request.params['title'] = vpath.pop(0) # /album title/
            return self.albums

        return vpath

    @cherrypy.expose
    def index(self, name):
        return 'About %s...' % name

class Album(object):
    @cherrypy.expose
    def index(self, artist, title):
        return 'About %s by %s...' % (title, artist)

if __name__ == '__main__':
    cherrypy.quickstart(Band())
```

Notice how the controller defines `_cp_dispatch`, it takes a single argument, the URL path info broken into its segments.

The method can inspect and manipulate the list of segments, removing any or adding new segments at any position. The new list of segments is then sent to the dispatcher which will use it to locate the appropriate resource.

In the above example, you should be able to go to the following URLs:

- http://localhost:8080/nirvana/
- http://localhost:8080/nirvana/albums/nevermind/

The `/nirvana/` segment is associated to the band and the `/nevermind/` segment relates to the album.

To achieve this, our `_cp_dispatch` method works on the idea that the default dispatcher matches URLs against page handler signatures and their position in the tree of handlers.

In this case, we take the dynamic segments in the URL (band and record names), we inject them into the request parameters and we remove them from the segment lists as if they had never been there in the first place.

In other words, `_cp_dispatch` makes it as if we were working on the following URLs:

- http://localhost:8080/?artist=nirvana
- http://localhost:8080/albums/?artist=nirvana&title=nevermind

5.2.2 The popargs decorator

cherrypy.popargs() is more straightforward as it gives a name to any segment that CherryPy wouldn't be able to interpret otherwise. This makes the matching of segments with page handler signatures easier and helps CherryPy understand the structure of your URL.

```
import cherrypy

@cherrypy.popargs('band_name')
class Band(object):
    def __init__(self):
        self.albums = Album()

    @cherrypy.expose
    def index(self, band_name):
        return 'About %s...' % band_name

@cherrypy.popargs('album_title')
class Album(object):
    @cherrypy.expose
    def index(self, band_name, album_title):
        return 'About %s by %s...' % (album_title, band_name)

if __name__ == '__main__':
    cherrypy.quickstart(Band())
```

This works similarly to `_cp_dispatch` but, as said above, is more explicit and localized. It says:

- take the first segment and store it into a parameter named `band_name`
- take again the first segment (since we removed the previous first) and store it into a parameter named `album_title`

Note that the decorator accepts more than a single binding. For instance:

```
@cherrypy.popargs('album_title')
class Album(object):
    def __init__(self):
        self.tracks = Track()

@cherrypy.popargs('track_num', 'track_title')
class Track(object):
    @cherrypy.expose
    def index(self, band_name, album_title, track_num, track_title):
        ...
```

This would handle the following URL:

- http://localhost:8080/nirvana/albums/nevermind/tracks/06/polly

Notice finally how the whole stack of segments is passed to each page handler so that you have the full context.

5.3 Error handling

CherryPy's `HTTPError` class supports raising immediate responses in the case of errors.

```
class Root:
    @cherrypy.expose
    def thing(self, path):
        if not authorized():
            raise cherrypy.HTTPError(401, 'Unauthorized')
        try:
            file = open(path)
        except FileNotFoundError:
            raise cherrypy.HTTPError(404)
```

`HTTPError.handle` is a context manager which supports translating exceptions raised in the app into an appropriate HTTP response, as in the second example.

```
class Root:
    @cherrypy.expose
    def thing(self, path):
        with cherrypy.HTTPError.handle(FileNotFoundError, 404):
            file = open(path)
```

5.4 Streaming the response body

CherryPy handles HTTP requests, packing and unpacking the low-level details, then passing control to your application's *page handler*, which produce the body of the response. CherryPy allows you to return body content in a variety of types: a string, a list of strings, a file. CherryPy also allows you to *yield* content, rather than *return* content. When you use "yield", you also have the option of streaming the output.

In general, it is safer and easier to not stream output. Therefore, streaming output is off by default. Streaming output and also using sessions requires a good understanding of *how session locks work*.

5.4.1 The "normal" CherryPy response process

When you provide content from your page handler, CherryPy manages the conversation between the HTTP server and your code like this:

Notice that the HTTP server gathers all output first and then writes everything to the client at once: status, headers, and body. This works well for static or simple pages, since the entire response can be changed at any time, either in your application code, or by the CherryPy framework.

5.4.2 How "streaming output" works with CherryPy

When you set the config entry "response.stream" to True (and use "yield"), CherryPy manages the conversation between the HTTP server and your code like this:

When you stream, your application doesn't immediately pass raw body content back to CherryPy or to the HTTP server. Instead, it passes back a generator. At that point, CherryPy finalizes the status and headers, **before** the generator has been consumed, or has produced any output. This is necessary to allow the HTTP server to send the headers and pieces of the body as they become available.

Once CherryPy has set the status and headers, it sends them to the HTTP server, which then writes them out to the client. From that point on, the CherryPy framework mostly steps out of the way, and the HTTP server essentially requests content directly from your application code (your page handler method).

Therefore, when streaming, if an error occurs within your page handler, CherryPy will not catch it–the HTTP server will catch it. Because the headers (and potentially some of the body) have already been written to the client, the server *cannot* know a safe means of handling the error, and will therefore simply close the connection (the current, builtin servers actually write out a short error message in the body, but this may be changed, and is not guaranteed behavior for all HTTP servers you might use with CherryPy).

In addition, you cannot manually modify the status or headers within your page handler if that handler method is a streaming generator, because the method will not be iterated over until after the headers have been written to the client. **This includes raising exceptions like HTTPError, NotFound, InternalRedirect and HTTPRedirect.** To use a streaming generator while modifying headers, you would have to return a generator that is separate from (or embedded in) your page handler. For example:

```
class Root:
    @cherrypy.expose
    def thing(self):
        cherrypy.response.headers['Content-Type'] = 'text/plain'
        if not authorized():
            raise cherrypy.NotFound()
        def content():
            yield "Hello, "
            yield "world"
        return content()
    thing._cp_config = {'response.stream': True}
```

Streaming generators are sexy, but they play havoc with HTTP. CherryPy allows you to stream output for specific situations: pages which take many minutes to produce, or pages which need a portion of their content immediately output to the client. Because of the issues outlined above, **it is usually better to flatten (buffer) content rather than stream content**. Do otherwise only when the benefits of streaming outweigh the risks.

5.5 Response timing

CherryPy responses include an attribute:

- `response.time`: the `time.time()` at which the response began

5.6 Deal with signals

This *engine plugin* is instantiated automatically as `cherrypy.engine.signal_handler`. However, it is only *subscribed* automatically by `cherrypy.quickstart()`. So if you want signal handling and you're calling:

```
tree.mount()
engine.start()
engine.block()
```

on your own, be sure to add before you start the engine:

```
engine.signals.subscribe()
```

5.6.1 Windows Console Events

Microsoft Windows uses console events to communicate some signals, like Ctrl-C. Deploying CherryPy on Windows platforms requires Python for Windows Extensions, which are installed automatically, being provided an extra dependency with environment marker. With that installed, CherryPy will handle Ctrl-C and other console events (CTRL_C_EVENT, CTRL_LOGOFF_EVENT, CTRL_BREAK_EVENT, CTRL_SHUTDOWN_EVENT, and CTRL_CLOSE_EVENT) automatically, shutting down the bus in preparation for process exit.

5.7 Securing your server

Note: This section is not meant as a complete guide to securing a web application or ecosystem. Please review the various guides provided at OWASP.

There are several settings that can be enabled to make CherryPy pages more secure. These include:

Transmitting data:

1. Use Secure Cookies

Rendering pages:

1. Set HttpOnly cookies
2. Set XFrame options
3. Enable XSS Protection
4. Set the Content Security Policy

An easy way to accomplish this is to set headers with a tool and wrap your entire CherryPy application with it:

```
import cherrypy

# set the priority according to your needs if you are hooking something
# else on the 'before_finalize' hook point.
@cherrypy.tools.register('before_finalize', priority=60)
def secureheaders():
    headers = cherrypy.response.headers
    headers['X-Frame-Options'] = 'DENY'
    headers['X-XSS-Protection'] = '1; mode=block'
    headers['Content-Security-Policy'] = "default-src 'self';"
```

Note: Read more about those headers.

Then, in the *configuration file* (or any other place that you want to enable the tool):

```
[/]
tools.secureheaders.on = True
```

If you use *sessions* you can also enable these settings:

```
[/]
tools.sessions.on = True
# increase security on sessions
tools.sessions.secure = True
tools.sessions.httponly = True
```

If you use SSL you can also enable Strict Transport Security:

```
 # add this to secureheaders():
 # only add Strict-Transport headers if we're actually using SSL; see the ietf spec
 # "An HSTS Host MUST NOT include the STS header field in HTTP responses
 # conveyed over non-secure transport"
 # http://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec-14#section-7.2
 if (cherrypy.server.ssl_certificate != None and cherrypy.server.ssl_private_key !=␣
↪None):
headers['Strict-Transport-Security'] = 'max-age=31536000'  # one year
```

Next, you should probably use *SSL*.

5.8 Multiple HTTP servers support

CherryPy starts its own HTTP server whenever you start the engine. In some cases, you may wish to host your application on more than a single port. This is easily achieved:

```
from cherrypy._cpserver import Server
server = Server()
server.socket_port = 8090
server.subscribe()
```

You can create as many `server` server instances as you need, once *subscribed*, they will follow the CherryPy engine's life-cycle.

5.9 WSGI support

CherryPy supports the WSGI interface defined in **PEP 333** as well as its updates in **PEP 3333**. It means the following:

- You can host a foreign WSGI application with the CherryPy server
- A CherryPy application can be hosted by another WSGI server

5.9.1 Make your CherryPy application a WSGI application

A WSGI application can be obtained from your application as follows:

```
import cherrypy
wsgiapp = cherrypy.Application(StringGenerator(), '/', config=myconf)
```

Simply use the *wsgiapp* instance in any WSGI-aware server.

5.9.2 Host a foreign WSGI application in CherryPy

Assuming you have a WSGI-aware application, you can host it in your CherryPy server using the *cherrypy.tree.graft* facility.

```
def raw_wsgi_app(environ, start_response):
    status = '200 OK'
    response_headers = [('Content-type','text/plain')]
    start_response(status, response_headers)
    return ['Hello world!']

cherrypy.tree.graft(raw_wsgi_app, '/')
```

Important: You cannot use tools with a foreign WSGI application. However, you can still benefit from the *CherryPy bus*.

5.9.3 No need for the WSGI interface?

The default CherryPy HTTP server supports the WSGI interfaces defined in **PEP 333** and **PEP 3333**. However, if your application is a pure CherryPy application, you can switch to a HTTP server that by-passes the WSGI layer altogether. It will provide a slight performance increase.

```
import cherrypy

class Root(object):
    @cherrypy.expose
    def index(self):
        return "Hello World!"

if __name__ == '__main__':
    from cherrypy._cpnative_server import CPHTTPServer
    cherrypy.server.httpserver = CPHTTPServer(cherrypy.server)

    cherrypy.quickstart(Root(), '/')
```

Important: Using the native server, you will not be able to graft a WSGI application as shown in the previous section. Doing so will result in a server error at runtime.

5.10 WebSocket support

WebSocket is a recent application protocol that came to life from the HTML5 working-group in response to the needs for bi-directional communication. Various hacks had been proposed such as Comet, polling, etc.

WebSocket is a socket that starts its life from a HTTP upgrade request. Once the upgrade is performed, the underlying socket is kept opened but not used in a HTTP context any longer. Instead, both connected endpoints may use the socket to push data to the other end.

CherryPy itself does not support WebSocket, but the feature is provided by an external library called ws4py.

5.11 Database support

CherryPy does not bundle any database access but its architecture makes it easy to integrate common database interfaces such as the DB-API specified in **PEP 249**. Alternatively, you can also use an ORM such as SQLAlchemy or SQLObject.

You will find a recipe at cherrypy-recipes that explains how to integrate SQLAlchemy using a mix of *plugins* and *tools*.

5.12 HTML Templating support

CherryPy does not provide any HTML template but its architecture makes it easy to integrate one. Popular ones are Mako or Jinja2.

You will find here a recipe on how to integrate them using a mix *plugins* and *tools*.

5.13 Testing your application

Web applications, like any other kind of code, must be tested. CherryPy provides a *helper class* to ease writing functional tests.

Here is a simple example for a basic echo application:

```
import cherrypy
from cherrypy.test import helper

class SimpleCPTest(helper.CPWebCase):
    def setup_server():
        class Root(object):
            @cherrypy.expose
            def echo(self, message):
                return message

        cherrypy.tree.mount(Root())
    setup_server = staticmethod(setup_server)

    def test_message_should_be_returned_as_is(self):
        self.getPage("/echo?message=Hello%20world")
        self.assertStatus('200 OK')
        self.assertHeader('Content-Type', 'text/html;charset=utf-8')
        self.assertBody('Hello world')
```

(continues on next page)

(continued from previous page)

```
    def test_non_utf8_message_will_fail(self):
        """
        CherryPy defaults to decode the query-string
        using UTF-8, trying to send a query-string with
        a different encoding will raise a 404 since
        it considers it's a different URL.
        """
        self.getPage("/echo?message=A+bient%F4t",
                     headers=[
                         ('Accept-Charset', 'ISO-8859-1,utf-8'),
                         ('Content-Type', 'text/html;charset=ISO-8859-1')
                     ]
        )
        self.assertStatus('404 Not Found')
```

As you can see the, test inherits from that helper class. You should setup your application and mount it as per-usual. Then, define your various tests and call the helper *`getPage()`* method to perform a request. Simply use the various specialized assert* methods to validate your workflow and data.

You can then run the test using py.test as follows:

```
$ py.test -s test_echo_app.py
```

The `-s` is necessary because the CherryPy class also wraps stdin and stdout. Without the flag, tests may hang on failed assertions waiting for an input.

Another option to avoid this problem, (if, for example, you are running tests inside an IDE) is to disable the interactive mode that's enabled by default. It can be disabled setting the `WEBTEST_INTERACTIVE` environment variable to `False` or `0`.

If you don't want to change environment variables to simply run a suite of tests you could also subclass the *`helper class`*, set `helper.CPWebCase.interactive = False` in the class and then derive all your test classes from your custom class:

```
import cherrypy
from cherrypy.test import helper

class TestsBase(helper.CPWebCase):

    helper.CPWebCase.interactive = False
```

Note: Although they are written using the typical pattern the `unittest` module supports, they are not bare unit tests. Indeed, a whole CherryPy stack is started for you and runs your application. If you want to really unit test your CherryPy application, meaning without having to start a server, you may want to have a look at this recipe.

Note: The *`helper class`* derives from `unittest.TestCase` class. For this reason, running from pytest, there are some limitations with respect to standard `pytest` tests, especially if you are grouping the tests in test classes. You can find more details at this page.

CHAPTER

SIX

CONFIGURE

Configuration in CherryPy is implemented via dictionaries. Keys are strings which name the mapped value; values may be of any type.

In CherryPy 3, you use configuration (files or dicts) to set attributes directly on the engine, server, request, response, and log objects. So the best way to know the full range of what's available in the config file is to simply import those objects and see what `help(obj)` tells you.

Note: If you are new to CherryPy, please refer first to the simpler *basic config* section first.

Contents

6.1 Architecture

The first thing you need to know about CherryPy 3's configuration is that it separates *global* config from *application* config. If you're deploying multiple *applications* at the same *site* (and more and more people are, as Python web apps are tending to decentralize), you need to be careful to separate the configurations, as well. There's only ever one "global config", but there is a separate "app config" for each app you deploy.

CherryPy *Requests* are part of an *Application*, which runs in a *global* context, and configuration data may apply to any of those three scopes. Let's look at each of those scopes in turn.

6.1.1 Global config

Global config entries apply everywhere, and are stored in *cherrypy.config*. This flat dict only holds global config data; that is, "site-wide" config entries which affect all mounted applications.

Global config is stored in the *cherrypy.config* dict, and you therefore update it by calling `cherrypy.config.update(conf)`. The `conf` argument can be either a filename, an open file, or a dict of config entries. Here's an example of passing a dict argument:

```
cherrypy.config.update({'server.socket_host': '64.72.221.48',
                        'server.socket_port': 80,
                       })
```

The `server.socket_host` option in this example determines on which network interface CherryPy will listen. The `server.socket_port` option declares the TCP port on which to listen.

6.1.2 Application config

Application entries apply to a single mounted application, and are stored on each Application object itself as *app.config*. This is a two-level dict where each top-level key is a path, or "relative URL" (for example, `"/"` or `"/my/page"`), and each value is a dict of config entries. The URL's are relative to the script name (mount point) of the Application. Usually, all this data is provided in the call to `tree.mount(root(), script_name='/path/to', config=conf)`, although you may also use `app.merge(conf)`. The `conf` argument can be either a filename, an open file, or a dict of config entries.

Configuration file example:

```
[/]
tools.trailing_slash.on = False
request.dispatch: cherrypy.dispatch.MethodDispatcher()
```

or, in python code:

```
config = {'/':
    {
        'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
        'tools.trailing_slash.on': False,
    }
}
cherrypy.tree.mount(Root(), config=config)
```

CherryPy only uses sections that start with `"/"` (except `[global]`, see below). That means you can place your own configuration entries in a CherryPy config file by giving them a section name which does not start with `"/"`. For example, you might include database entries like this:

```
[global]
server.socket_host: "0.0.0.0"

[Databases]
driver: "postgres"
host: "localhost"
port: 5432

[/path]
response.timeout: 6000
```

Then, in your application code you can read these values during request time via `cherrypy.request.app.config['Databases']`. For code that is outside the request process, you'll have to pass a reference to your Application around.

6.1.3 Request config

Each Request object possesses a single *`request.config`* dict. Early in the request process, this dict is populated by merging Global config, Application config, and any config acquired while looking up the page handler (see next). This dict contains only those config entries which apply to the given request.

Note: when you do an *`InternalRedirect`*, this config attribute is recalculated for the new path.

6.2 Declaration

Configuration data may be supplied as a Python dictionary, as a filename, or as an open file object.

6.2.1 Configuration files

When you supply a filename or file, CherryPy uses Python's builtin ConfigParser; you declare Application config by writing each path as a section header, and each entry as a `"key: value"` (or `"key = value"`) pair:

```
[/path/to/my/page]
response.stream: True
tools.trailing_slash.extra = False
```

Combined Configuration Files

If you are only deploying a single application, you can make a single config file that contains both global and app entries. Just stick the global entries into a config section named `[global]`, and pass the same file to both *`config.update`* and `tree.mount <cherrypy._cptree.Tree.mount()`. If you're calling `cherrypy.quickstart(app root, script name, config)`, it will pass the config to both places for you. But as soon as you decide to add another application to the same site, you need to separate the two config files/dicts.

Separate Configuration Files

If you're deploying more than one application in the same process, you need (1) file for global config, plus (1) file for *each* Application. The global config is applied by calling *cherrypy.config.update*, and application config is usually passed in a call to *cherrypy.tree.mount*.

In general, you should set global config first, and then mount each application with its own config. Among other benefits, this allows you to set up global logging so that, if something goes wrong while trying to mount an application, you'll see the tracebacks. In other words, use this order:

```
# global config
cherrypy.config.update({'environment': 'production',
                        'log.error_file': 'site.log',
                        # ...
                        })

# Mount each app and pass it its own config
cherrypy.tree.mount(root1, "", appconf1)
cherrypy.tree.mount(root2, "/forum", appconf2)
cherrypy.tree.mount(root3, "/blog", appconf3)

if hasattr(cherrypy.engine, 'block'):
    # 3.1 syntax
    cherrypy.engine.start()
    cherrypy.engine.block()
else:
    # 3.0 syntax
    cherrypy.server.quickstart()
    cherrypy.engine.start()
```

Values in config files use Python syntax

Config entries are always a key/value pair, like `server.socket_port = 8080`. The key is always a name, and the value is always a Python object. That is, if the value you are setting is an `int` (or other number), it needs to look like a Python `int`; for example, `8080`. If the value is a string, it needs to be quoted, just like a Python string. Arbitrary objects can also be created, just like in Python code (assuming they can be found/imported). Here's an extended example, showing you some of the different types:

```
[global]
log.error_file: "/home/fumanchu/myapp.log"
environment = 'production'
server.max_request_body_size: 1200

[/myapp]
tools.trailing_slash.on = False
request.dispatch: cherrypy.dispatch.MethodDispatcher()
```

6.2.2 _cp_config: attaching config to handlers

Config files have a severe limitation: values are always keyed by URL. For example:

```
[/path/to/page]
methods_with_bodies = ("POST", "PUT", "PROPPATCH")
```

It's obvious that the extra method is the norm for that path; in fact, the code could be considered broken without it. In CherryPy, you can attach that bit of config directly on the page handler:

```
@cherrypy.expose
def page(self):
    return "Hello, world!"
page._cp_config = {"request.methods_with_bodies": ("POST", "PUT", "PROPPATCH")}
```

`_cp_config` is a reserved attribute which the dispatcher looks for at each node in the object tree. The `_cp_config` attribute must be a CherryPy config dictionary. If the dispatcher finds a `_cp_config` attribute, it merges that dictionary into the rest of the config. The entire merged config dictionary is placed in *`cherrypy.request.config`*.

This can be done at any point in the tree of objects; for example, we could have attached that config to a class which contains the page method:

```
class SetOPages:

    _cp_config = {"request.methods_with_bodies": ("POST", "PUT", "PROPPATCH")}

    @cherrypy.expose
    def page(self):
        return "Hullo, Werld!"
```

Note: This behavior is only guaranteed for the default dispatcher. Other dispatchers may have different restrictions on where you can attach `_cp_config` attributes. Additionally, because the dispatcher is is responsible for processing `_cp_config`, it is not possible to change the dispatcher (i.e. `request.dispatch` is not honored at this construct).

This technique allows you to:

- Put config near where it's used for improved readability and maintainability.
- Attach config to objects instead of URL's. This allows multiple URL's to point to the same object, yet you only need to define the config once.
- Provide defaults which are still overridable in a config file.

6.3 Namespaces

Because config entries usually just set attributes on objects, they're almost all of the form: `object.attribute`. A few are of the form: `object.subobject.attribute`. They look like normal Python attribute chains, because they work like them. We call the first name in the chain the *"config namespace"*. When you provide a config entry, it is bound as early as possible to the actual object referenced by the namespace; for example, the entry `response.stream` actually sets the `stream` attribute of *`cherrypy.response`*! In this way, you can easily determine the default value by firing up a python interpreter and typing:

```
>>> import cherrypy
>>> cherrypy.response.stream
False
```

Each config namespace has its own handler; for example, the "request" namespace has a handler which takes your config entry and sets that value on the appropriate "request" attribute. There are a few namespaces, however, which don't work like normal attributes behind the scenes; however, they still use dotted keys and are considered to "have a namespace".

6.3.1 Builtin namespaces

Entries from each namespace may be allowed in the global, application root (`"/"`) or per-path config, or a combination:

Scope	Global	Application Root	App Path
engine	X		
hooks	X	X	X
log	X	X	
request	X	X	X
response	X	X	X
server	X		
tools	X	X	X

engine

Entries in this namespace controls the 'application engine'. These can only be declared in the global config. Any attribute of *cherrypy.engine* may be set in config; however, there are a few extra entries available in config:

- Plugin attributes. Many of the Engine Plugins are themselves attributes of `cherrypy.engine`. You can set any attribute of an attached plugin by simply naming it. For example, there is an instance of the *Autoreloader* class at `engine.autoreload`; you can set its "frequency" attribute via the config entry `engine.autoreload.frequency = 60`. In addition, you can turn such plugins on and off by setting `engine.autoreload.on = True` or `False`.
- `engine.SIGHUP/SIGTERM`: These entries can be used to set the list of listeners for the given channel. Mostly, this is used to turn off the signal handling one gets automatically via *cherrypy.quickstart()*.

hooks

Declares additional request-processing functions. Use this to append your own *Hook* functions to the request. For example, to add `my_hook_func` to the `before_handler` hookpoint:

```
[/]
hooks.before_handler = myapp.my_hook_func
```

log

Configures logging. These can only be declared in the global config (for global logging) or `[/]` config (for each application). See *LogManager* for the list of configurable attributes. Typically, the "access_file", "error_file", and "screen" attributes are the most commonly configured.

request

Sets attributes on each Request. See the *Request* class for a complete list.

response

Sets attributes on each Response. See the *Response* class for a complete list.

server

Controls the default HTTP server via *cherrypy.server* (see that class for a complete list of configurable attributes). These can only be declared in the global config.

tools

Enables and configures additional request-processing packages. See the /tutorial/tools overview for more information.

wsgi

Adds WSGI middleware to an Application's "pipeline". These can only be declared in the app's root config ("/").

- `wsgi.pipeline`: Appends to the WSGi pipeline. The value must be a list of (name, app factory) pairs. Each app factory must be a WSGI callable class (or callable that returns a WSGI callable); it must take an initial 'nextapp' argument, plus any optional keyword arguments. The optional arguments may be configured via `wsgi.<name>.<arg>`.
- `wsgi.response_class`: Overrides the default *Response* class.

checker

Controls the "checker", which looks for common errors in app state (including config) when the engine starts. You can turn off individual checks by setting them to `False` in config. See *cherrypy._cpchecker.Checker* for a complete list. Global config only.

6.3.2 Custom config namespaces

You can define your own namespaces if you like, and they can do far more than simply set attributes. The `test/test_config` module, for example, shows an example of a custom namespace that coerces incoming params and outgoing body content. The *cherrypy._cpwsgi* module includes an additional, builtin namespace for invoking WSGI middleware.

In essence, a config namespace handler is just a function, that gets passed any config entries in its namespace. You add it to a namespaces registry (a dict), where keys are namespace names and values are handler functions. When a config entry for your namespace is encountered, the corresponding handler function will be called, passing the config key and value; that is, `namespaces[namespace](k, v)`. For example, if you write:

```
def db_namespace(k, v):
    if k == 'connstring':
        orm.connect(v)
cherrypy.config.namespaces['db'] = db_namespace
```

then `cherrypy.config.update({"db.connstring": "Oracle:host=1.10.100.200; sid=TEST"})` will call `db_namespace('connstring', 'Oracle:host=1.10.100.200; sid=TEST')`.

The point at which your namespace handler is called depends on where you add it:

Scope	Namespace dict	Handler is called in
Global	`cherrypy.config.namespaces`	cherrypy.config.update
Application	*`app.namespaces`*	Application.merge (which is called by cherrypy.tree.mount)
Request	*`app.request_class.namespaces`*	Request.configure (called for each request, after the handler is looked up)

The name can be any string, and the handler must be either a callable or a (Python 2.5 style) context manager.

If you need additional code to run when all your namespace keys are collected, you can supply a callable context manager in place of a normal function for the handler. Context managers are defined in **PEP 343**.

6.3.3 Environments

The only key that does not exist in a namespace is the *"environment"* entry. It only applies to the global config, and only when you use *`cherrypy.config.update`*. This special entry *imports* other config entries from the following template stored in `cherrypy._cpconfig.environments[environment]`.

```
Config.environments = environments = {
    'staging': {
        'engine.autoreload.on': False,
        'checker.on': False,
        'tools.log_headers.on': False,
        'request.show_tracebacks': False,
        'request.show_mismatched_params': False,
    },
    'production': {
        'engine.autoreload.on': False,
        'checker.on': False,
        'tools.log_headers.on': False,
        'request.show_tracebacks': False,
        'request.show_mismatched_params': False,
        'log.screen': False,
    },
    'embedded': {
        # For use with CherryPy embedded in another deployment stack.
        'engine.autoreload.on': False,
        'checker.on': False,
        'tools.log_headers.on': False,
        'request.show_tracebacks': False,
        'request.show_mismatched_params': False,
        'log.screen': False,
        'engine.SIGHUP': None,
        'engine.SIGTERM': None,
    },
    'test_suite': {
        'engine.autoreload.on': False,
        'checker.on': False,
```

(continues on next page)

(continued from previous page)

```
        'tools.log_headers.on': False,
        'request.show_tracebacks': True,
        'request.show_mismatched_params': True,
        'log.screen': False,
    },
}
```

If you find the set of existing environments (production, staging, etc) too limiting or just plain wrong, feel free to extend them or add new environments:

```
cherrypy._cpconfig.environments['staging']['log.screen'] = False

cherrypy._cpconfig.environments['Greek'] = {
    'tools.encode.encoding': 'ISO-8859-7',
    'tools.decode.encoding': 'ISO-8859-7',
    }
```

CHAPTER

SEVEN

EXTEND

CherryPy is truly an open framework, you can extend and plug new functions at will either server-side or on a per-requests basis. Either way, CherryPy is made to help you build your application and support your architecture via simple patterns.

Contents

7.1 Server-wide functions

CherryPy can be considered both as a HTTP library as much as a web application framework. In that latter case, its architecture provides mechanisms to support operations across the whole server instance. This offers a powerful canvas to perform persistent operations as server-wide functions live outside the request processing itself. They are available to the whole process as long as the bus lives.

Typical use cases:

- Keeping a pool of connection to an external server so that your need not to re-open them on each request (database connections for instance).
- Background processing (say you need work to be done without blocking the whole request itself).

7.1.1 Publish/Subscribe pattern

CherryPy's backbone consists of a bus system implementing a simple publish/subscribe messaging pattern. Simply put, in CherryPy everything is controlled via that bus. One can easily picture the bus as a sushi restaurant's belt as in the picture below.

You can subscribe and publish to channels on a bus. A channel is bit like a unique identifier within the bus. When a message is published to a channel, the bus will dispatch the message to all subscribers for that channel.

One interesting aspect of a pubsub pattern is that it promotes decoupling between a caller and the callee. A published message will eventually generate a response but the publisher does not know where that response came from.

Thanks to that decoupling, a CherryPy application can easily access functionalities without having to hold a reference to the entity providing that functionality. Instead, the application simply publishes onto the bus and will receive the appropriate response, which is all that matter.

Typical pattern

Let's take the following dummy application:

```
import cherrypy

class ECommerce(object):
    def __init__(self, db):
        self.mydb = db

    @cherrypy.expose
    def save_kart(self, cart_data):
        cart = Cart(cart_data)
        self.mydb.save(cart)

if __name__ == '__main__':
   cherrypy.quickstart(ECommerce(), '/')
```

The application has a reference to the database but this creates a fairly strong coupling between the database provider and the application.

Another approach to work around the coupling is by using a pubsub workflow:

```
import cherrypy

class ECommerce(object):
    @cherrypy.expose
    def save_kart(self, cart_data):
        cart = Cart(cart_data)
        cherrypy.engine.publish('db-save', cart)

if __name__ == '__main__':
   cherrypy.quickstart(ECommerce(), '/')
```

In this example, we publish a `cart` instance to `db-save` channel. One or many subscribers can then react to that message and the application doesn't have to know about them.

Note: This approach is not mandatory and it's up to you to decide how to design your entities interaction.

Implementation details

CherryPy's bus implementation is simplistic as it registers functions to channels. Whenever a message is published to a channel, each registered function is applied with that message passed as a parameter.

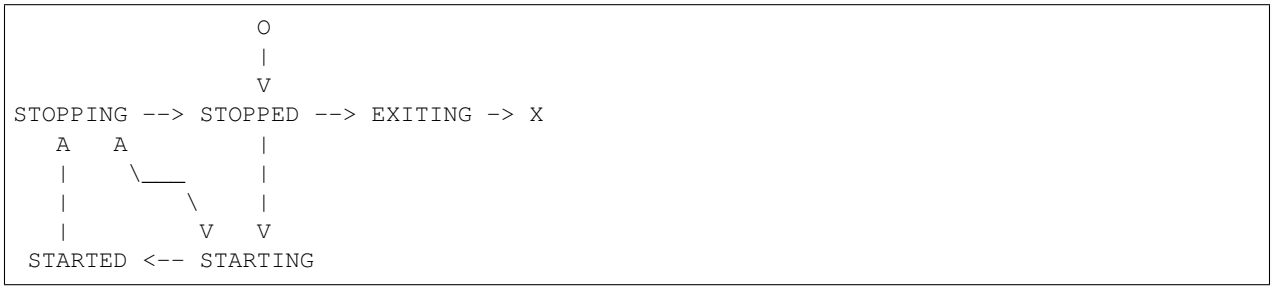
The whole behaviour happens synchronously and, in that sense, if a subscriber takes too long to process a message, the remaining subscribers will be delayed.

CherryPy's bus is not an advanced pubsub messaging broker system such as provided by zeromq or RabbitMQ. Use it with the understanding that it may have a cost.

Engine as a pubsub bus

As said earlier, CherryPy is built around a pubsub bus. All entities that the framework manages at runtime are working on top of a single bus instance, which is named the `engine`.

The bus implementation therefore provides a set of common channels which describe the application's lifecycle:



The states' transitions trigger channels to be published to so that subscribers can react to them.

One good example is the HTTP server which will tranisition from a `"STOPPED"` stated to a `"STARTED"` state whenever a message is published to the *`start`* channel.

Built-in channels

In order to support its life-cycle, CherryPy defines a set of common channels that will be published to at various states:

- **"start"**: When the bus is in the `"STARTING"` state
- **"main"**: Periodically from the CherryPy's mainloop
- **"stop"**: When the bus is in the `"STOPPING"` state
- **"graceful"**: When the bus requests a reload of subscribers
- **"exit"**: When the bus is in the `"EXITING"` state

This channel will be published to by the `engine` automatically. Register therefore any subscribers that would need to react to the transition changes of the `engine`.

In addition, a few other channels are also published to during the request processing.

- **"before_request"**: right before the request is processed by CherryPy
- **"after_request"**: right after it has been processed

Also, from the *`cherrypy.process.plugins.ThreadManager`* plugin:

- **"acquire_thread"**
- **"start_thread"**
- **"stop_thread"**
- **"release_thread"**

Bus API

In order to work with the bus, the implementation provides the following simple API:

- *cherrypy.engine.publish(channel, *args)*:
- The `channel` parameter is a string identifying the channel to which the message should be sent to
- `*args` is the message and may contain any valid Python values or objects.
- *cherrypy.engine.subscribe(channel, callable)*:
- The `channel` parameter is a string identifying the channel the *callable* will be registered to.
- *callable* is a Python function or method which signature must match what will be published.
- *cherrypy.engine.unsubscribe(channel, callable)*:
- The `channel` parameter is a string identifying the channel the *callable* was registered to.
- *callable* is the Python function or method which was registered.

7.1.2 Plugins

Plugins, simply put, are entities that play with the bus, either by publishing or subscribing to channels, usually both at the same time.

Important: Plugins are extremely useful whenever you have functionalities:

- Available across the whole application server
- Associated to the application's life-cycle
- You want to avoid being strongly coupled to the application

---

Create a plugin

A typical plugin looks like this:

```
import cherrypy
from cherrypy.process import wspbus, plugins

class DatabasePlugin(plugins.SimplePlugin):
    def __init__(self, bus, db_klass):
        plugins.SimplePlugin.__init__(self, bus)
        self.db = db_klass()

    def start(self):
        self.bus.log('Starting up DB access')
        self.bus.subscribe("db-save", self.save_it)

    def stop(self):
        self.bus.log('Stopping down DB access')
        self.bus.unsubscribe("db-save", self.save_it)

    def save_it(self, entity):
        self.db.save(entity)
```

The *cherrypy.process.plugins.SimplePlugin* is a helper class provided by CherryPy that will automatically subscribe your *start* and *stop* methods to the related channels.

When the *start* and *stop* channels are published on, those methods are called accordingly.

Notice then how our plugin subscribes to the `db-save` channel so that the bus can dispatch messages to the plugin.

Enable a plugin

To enable the plugin, it has to be registered to the the bus as follows:

```
DatabasePlugin(cherrypy.engine, SQLiteDB).subscribe()
```

The `SQLiteDB` here is a fake class that is used as our database provider.

Disable a plugin

You can also unregister a plugin as follows:

```
someplugin.unsubscribe()
```

This is often used when you want to prevent the default HTTP server from being started by CherryPy, for instance if you run on top of a different HTTP server (WSGI capable):

```
cherrypy.server.unsubscribe()
```

Let's see an example using this default application:

```
import cherrypy

class Root(object):
    @cherrypy.expose
    def index(self):
        return "hello world"

if __name__ == '__main__':
    cherrypy.quickstart(Root())
```

For instance, this is what you would see when running this application:

```
[27/Apr/2014:13:04:07] ENGINE Listening for SIGHUP.
[27/Apr/2014:13:04:07] ENGINE Listening for SIGTERM.
[27/Apr/2014:13:04:07] ENGINE Listening for SIGUSR1.
[27/Apr/2014:13:04:07] ENGINE Bus STARTING
[27/Apr/2014:13:04:07] ENGINE Started monitor thread 'Autoreloader'.
[27/Apr/2014:13:04:08] ENGINE Serving on http://127.0.0.1:8080
[27/Apr/2014:13:04:08] ENGINE Bus STARTED
```

Now let's unsubscribe the HTTP server:

```
import cherrypy

class Root(object):
    @cherrypy.expose
    def index(self):
        return "hello world"
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    cherrypy.server.unsubscribe()
    cherrypy.quickstart(Root())
```

This is what we get:

```
[27/Apr/2014:13:08:06] ENGINE Listening for SIGHUP.
[27/Apr/2014:13:08:06] ENGINE Listening for SIGTERM.
[27/Apr/2014:13:08:06] ENGINE Listening for SIGUSR1.
[27/Apr/2014:13:08:06] ENGINE Bus STARTING
[27/Apr/2014:13:08:06] ENGINE Started monitor thread 'Autoreloader'.
[27/Apr/2014:13:08:06] ENGINE Bus STARTED
```

As you can see, the server is not started. The missing:

```
[27/Apr/2014:13:04:08] ENGINE Serving on http://127.0.0.1:8080
```

7.2 Per-request functions

One of the most common task in a web application development is to tailor the request's processing to the runtime context.

Within CherryPy, this is performed via what are called *Tools*. If you are familiar with Django or WSGI middlewares, CherryPy tools are similar in spirit. They add functions that are applied during the request/response processing.

7.2.1 Hook point

A hook point is a point during the request/response processing.

Here is a quick rundown of the "hook points" that you can hang your tools on:

- **"on_start_resource"** - The earliest hook; the Request-Line and request headers have been processed and a dispatcher has set request.handler and request.config.
- **"before_request_body"** - Tools that are hooked up here run right before the request body would be processed.
- **"before_handler"** - Right before the request.handler (the *exposed* callable that was found by the dispatcher) is called.
- **"before_finalize"** - This hook is called right after the page handler has been processed and before CherryPy formats the final response object. It helps you for example to check for what could have been returned by your page handler and change some headers if needed.
- **"on_end_resource"** - Processing is complete - the response is ready to be returned. This doesn't always mean that the request.handler (the exposed page handler) has executed! It may be a generator. If your tool absolutely needs to run after the page handler has produced the response body, you need to either use on_end_request instead, or wrap the response.body in a generator which applies your tool as the response body is being generated.
- **"before_error_response"** - Called right before an error response (status code, body) is set.
- **"after_error_response"** - Called right after the error response (status code, body) is set and just before the error response is finalized.
- **"on_end_request"** - The request/response conversation is over, all data has been written to the client, nothing more to see here, move along.

7.2.2 Tools

A tool is a simple callable object (function, method, object implementing a `__call__` method) that is attached to a *hook point*.

Below is a simple tool that is attached to the `before_finalize` hook point, hence after the page handler was called:

```
@cherrypy.tools.register('before_finalize')
def logit():
   print(cherrypy.request.remote.ip)
```

Tools can also be created and assigned manually. The decorator registration is equivalent to:

```
cherrypy.tools.logit = cherrypy.Tool('before_finalize', logit)
```

Using that tool is as simple as follows:

```
class Root(object):
    @cherrypy.expose
    @cherrypy.tools.logit()
    def index(self):
        return "hello world"
```

Obviously the tool may be declared the *other usual ways*.

Note: The name of the tool, technically the attribute set to `cherrypy.tools`, does not have to match the name of the callable. However, it is that name that will be used in the configuration to refer to that tool.

Stateful tools

The tools mechanism is really flexible and enables rich per-request functionalities.

Straight tools as shown in the previous section are usually good enough. However, if your workflow requires some sort of state during the request processing, you will probably want a class-based approach:

```
import time

import cherrypy

class TimingTool(cherrypy.Tool):
    def __init__(self):
        cherrypy.Tool.__init__(self, 'before_handler',
                               self.start_timer,
                               priority=95)

    def _setup(self):
        cherrypy.Tool._setup(self)
        cherrypy.request.hooks.attach('before_finalize',
                                      self.end_timer,
                                      priority=5)

    def start_timer(self):
        cherrypy.request._time = time.time()
```

(continues on next page)

(continued from previous page)

```
    def end_timer(self):
        duration = time.time() - cherrypy.request._time
        cherrypy.log("Page handler took %.4f" % duration)

cherrypy.tools.timeit = TimingTool()
```

This tool computes the time taken by the page handler for a given request. It stores the time at which the handler is about to get called and logs the time difference right after the handler returned its result.

The import bits is that the *cherrypy.Tool* constructor allows you to register to a hook point but, to attach the same tool to a different hook point, you must use the *cherrypy.request.hooks.attach* method. The *cherrypy.Tool._setup* method is automatically called by CherryPy when the tool is applied to the request.

Next, let's see how to use our tool:

```
class Root(object):
    @cherrypy.expose
    @cherrypy.tools.timeit()
    def index(self):
        return "hello world"
```

Tools ordering

Since you can register many tools at the same hookpoint, you may wonder in which order they will be applied.

CherryPy offers a deterministic, yet so simple, mechanism to do so. Simply set the **priority** attribute to a value from 1 to 100, lower values providing greater priority.

If you set the same priority for several tools, they will be called in the order you declare them in your configuration.

Toolboxes

All of the builtin CherryPy tools are collected into a Toolbox called `cherrypy.tools`. It responds to config entries in the `"tools"` *namespace*. You can add your own Tools to this Toolbox as described above.

You can also make your own Toolboxes if you need more modularity. For example, you might create multiple Tools for working with JSON, or you might publish a set of Tools covering authentication and authorization from which everyone could benefit (hint, hint). Creating a new Toolbox is as simple as:

```
import cherrypy

# Create a new Toolbox.
newauthtools = cherrypy._cptools.Toolbox("newauth")

# Add a Tool to our new Toolbox.
@newauthtools.register('before_request_body')
def check_access(default=False):
    if not getattr(cherrypy.request, "userid", default):
        raise cherrypy.HTTPError(401)
```

Then, in your application, use it just like you would use `cherrypy.tools`, with the additional step of registering your toolbox with your app. Note that doing so automatically registers the `"newauth"` config namespace; you can see the config entries in action below:

```
import cherrypy

class Root(object):
    @cherrypy.expose
    def default(self):
        return "Hello"

conf = {
   '/demo': {
       'newauth.check_access.on': True,
       'newauth.check_access.default': True,
    }
}

app = cherrypy.tree.mount(Root(), config=conf)
```

7.2.3 Request parameters manipulation

HTTP uses strings to carry data between two endpoints. However your application may make better use of richer object types. As it wouldn't be really readable, nor a good idea regarding maintenance, to let each page handler deserialize data, it's a common pattern to delegate this functions to tools.

For instance, let's assume you have a user id in the query-string and some user data stored into a database. You could retrieve the data, create an object and pass it on to the page handler instead of the user id.

```
import cherrypy

class UserManager(cherrypy.Tool):
    def __init__(self):
        cherrypy.Tool.__init__(self, 'before_handler',
                               self.load, priority=10)

    def load(self):
        req = cherrypy.request

        # let's assume we have a db session
        # attached to the request somehow
        db = req.db

        # retrieve the user id and remove it
        # from the request parameters
        user_id = req.params.pop('user_id')
        req.params['user'] = db.get(int(user_id))

cherrypy.tools.user = UserManager()


class Root(object):
    @cherrypy.expose
    @cherrypy.tools.user()
    def index(self, user):
        return "hello %s" % user.name
```

In other words, CherryPy give you the power to:

- inject data, that wasn't part of the initial request, into the page handler

- remove data as well
- convert data into a different, more useful, object to remove that burden from the page handler itself

7.3 Tailored dispatchers

Dispatching is the art of locating the appropriate page handler for a given request. Usually, dispatching is based on the request's URL, the query-string and, sometimes, the request's method (GET, POST, etc.).

Based on this, CherryPy comes with various dispatchers already.

In some cases however, you will need a little more. Here is an example of dispatcher that will always ensure the incoming URL leads to a lower-case page handler.

```
import random
import string

import cherrypy
from cherrypy._cpdispatch import Dispatcher

class StringGenerator(object):
   @cherrypy.expose
   def generate(self, length=8):
       return ''.join(random.sample(string.hexdigits, int(length)))

class ForceLowerDispatcher(Dispatcher):
    def __call__(self, path_info):
        return Dispatcher.__call__(self, path_info.lower())

if __name__ == '__main__':
    conf = {
        '/': {
            'request.dispatch': ForceLowerDispatcher(),
        }
    }
    cherrypy.quickstart(StringGenerator(), '/', conf)
```

Once you run this snippet, go to:

- http://localhost:8080/generate?length=8
- http://localhost:8080/GENerAte?length=8

In both cases, you will be led to the `generate` page handler. Without our home-made dispatcher, the second one would fail and return a 404 error (**RFC 7231#section-6.5.4**).

7.3.1 Tool or dispatcher?

In the previous example, why not simply use a tool? Well, the sooner a tool can be called is always after the page handler has been found. In our example, it would be already too late as the default dispatcher would have not even found a match for `/GENerAte`.

A dispatcher exists mostly to determine the best page handler to serve the requested resource.

On the other hand, tools are there to adapt the request's processing to the runtime context of the application and the request's content.

Usually, you will have to write a dispatcher only if you have a very specific use case to locate the most adequate page handler. Otherwise, the default ones will likely suffice.

7.4 Request body processors

Since its 3.2 release, CherryPy provides a really elegant and powerful mechanism to deal with a request's body based on its mimetype. Refer to the *cherrypy._cpreqbody* module to understand how to implement your own processors.

CHAPTER

EIGHT

DEPLOY

CherryPy stands on its own, but as an application server, it is often located in shared or complex environments. For this reason, it is not uncommon to run CherryPy behind a reverse proxy or use other servers to host the application.

Note: CherryPy's server has proven reliable and fast enough for years now. If the volume of traffic you receive is average, it will do well enough on its own. Nonetheless, it is common to delegate the serving of static content to more capable servers such as nginx or CDN.

Contents

8.1 Run as a daemon

CherryPy allows you to easily decouple the current process from the parent environment, using the traditional double-fork:

```
from cherrypy.process.plugins import Daemonizer
d = Daemonizer(cherrypy.engine)
d.subscribe()
```

Note: This *engine plugin* is only available on Unix and similar systems which provide `fork()`.

If a startup error occurs in the forked children, the return code from the parent process will still be 0. Errors in the initial daemonizing process still return proper exit codes, but errors after the fork won't. Therefore, if you use this plugin to daemonize, don't use the return code as an accurate indicator of whether the process fully started. In fact, that return code only indicates if the process successfully finished the first fork.

The plugin takes optional arguments to redirect standard streams: `stdin`, `stdout`, and `stderr`. By default, these are all redirected to `/dev/null`, but you're free to send them to log files or elsewhere.

Warning: You should be careful to not start any threads before this plugin runs. The plugin will warn if you do so, because "...the effects of calling functions that require certain resources between the call to fork() and the call to an exec function are undefined". (ref). It is for this reason that the Server plugin runs at priority 75 (it starts worker threads), which is later than the default priority of 65 for the Daemonizer.

8.2 Run as a different user

Use this *engine plugin* to start your CherryPy site as root (for example, to listen on a privileged port like 80) and then reduce privileges to something more restricted.

This priority of this plugin's "start" listener is slightly higher than the priority for `server.start` in order to facilitate the most common use: starting on a low port (which requires root) and then dropping to another user.

```
DropPrivileges(cherrypy.engine, uid=1000, gid=1000).subscribe()
```

8.3 PID files

The PIDFile *engine plugin* is pretty straightforward: it writes the process id to a file on start, and deletes the file on exit. You must provide a 'pidfile' argument, preferably an absolute path:

```
PIDFile(cherrypy.engine, '/var/run/myapp.pid').subscribe()
```

8.4 Systemd socket activation

Socket Activation is a systemd feature that allows to setup a system so that the systemd will sit on a port and start services 'on demand' (a little bit like inetd and xinetd used to do).

CherryPy has built-in socket activation support, if run from a systemd service file it will detect the `LISTEN_PID` environment variable to know that it should consider fd 3 to be the passed socket.

To read more about socket activation: http://0pointer.de/blog/projects/socket-activation.html

8.5 Control via Supervisord

Supervisord is a powerful process control and management tool that can perform a lot of tasks around process monitoring.

Below is a simple supervisor configuration for your CherryPy application.

```
[unix_http_server]
file=/tmp/supervisor.sock

[supervisord]
logfile=/tmp/supervisord.log ; (main log file;default $CWD/supervisord.log)
logfile_maxbytes=50MB        ; (max main logfile bytes b4 rotation;default 50MB)
logfile_backups=10           ; (num of main logfile rotation backups;default 10)
loglevel=info                ; (log level;default info; others: debug,warn,trace)
pidfile=/tmp/supervisord.pid ; (supervisord pidfile;default supervisord.pid)
nodaemon=false               ; (start in foreground if true;default false)
minfds=1024                  ; (min. avail startup file descriptors;default 1024)
minprocs=200                 ; (min. avail process descriptors;default 200)

[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface

[supervisorctl]
serverurl=unix:///tmp/supervisor.sock

[program:myapp]
command=python server.py
environment=PYTHONPATH=.
directory=.
```

This could control your server via the `server.py` module as the application entry point.

```
import cherrypy

class Root(object):
    @cherrypy.expose
    def index(self):
        return "Hello World!"


cherrypy.config.update({'server.socket_port': 8090,
                        'engine.autoreload.on': False,
                        'log.access_file': './access.log',
                        'log.error_file': './error.log'})
cherrypy.quickstart(Root())
```

To take the configuration (assuming it was saved in a file called `supervisor.conf`) into account:

```
$ supervisord -c supervisord.conf
$ supervisorctl update
```

Now, you can point your browser at http://localhost:8090/ and it will display `Hello World!`.

To stop supervisor, type:

```
$ supervisorctl shutdown
```

This will obviously shutdown your application.

8.6 SSL support

Note: You may want to test your server for SSL using the services from Qualys, Inc.

CherryPy can encrypt connections using SSL to create an https connection. This keeps your web traffic secure. Here's how.

1. Generate a private key. We'll use openssl and follow the OpenSSL Keys HOWTO.:

```
$ openssl genrsa -out privkey.pem 2048
```

You can create either a key that requires a password to use, or one without a password. Protecting your private key with a password is much more secure, but requires that you enter the password every time you use the key. For example, you may have to enter the password when you start or restart your CherryPy server. This may or may not be feasible, depending on your setup.

If you want to require a password, add one of the `-aes128`, `-aes192` or `-aes256` switches to the command above. You should not use any of the DES, 3DES, or SEED algorithms to protect your password, as they are insecure.

SSL Labs recommends using 2048-bit RSA keys for security (see references section at the end).

2. Generate a certificate. We'll use openssl and follow the OpenSSL Certificates HOWTO. Let's start off with a self-signed certificate for testing:

```
$ openssl req -new -x509 -days 365 -key privkey.pem -out cert.pem
```

openssl will then ask you a series of questions. You can enter whatever values are applicable, or leave most fields blank. The one field you *must* fill in is the 'Common Name': enter the hostname you will use to access your site. If you are just creating a certificate to test on your own machine and you access the server by typing 'localhost' into your browser, enter the Common Name 'localhost'.

3. Decide whether you want to use python's built-in SSL library, or the pyOpenSSL library. CherryPy supports either.

    a) *Built-in.* To use python's built-in SSL, add the following line to your CherryPy config:

    ```
    cherrypy.server.ssl_module = 'builtin'
    ```

    b) *pyOpenSSL.* Because python did not have a built-in SSL library when CherryPy was first created, the default setting is to use pyOpenSSL. To use it you'll need to install it (we could recommend you install cython first):

```
$ pip install cython, pyOpenSSL
```

4. Add the following lines in your CherryPy config to point to your certificate files:

```
cherrypy.server.ssl_certificate = "cert.pem"
cherrypy.server.ssl_private_key = "privkey.pem"
```

5. If you have a certificate chain at hand, you can also specify it:

```
cherrypy.server.ssl_certificate_chain = "certchain.perm"
```

6. Start your CherryPy server normally. Note that if you are debugging locally and/or using a self-signed certificate, your browser may show you security warnings.

8.7 WSGI servers

8.7.1 Embedding into another WSGI framework

Though CherryPy comes with a very reliable and fast enough HTTP server, you may wish to integrate your CherryPy application within a different framework. To do so, we will benefit from the WSGI interface defined in **PEP 333** and **PEP 3333**.

Note that you should follow some basic rules when embedding CherryPy in a third-party WSGI server:

- If you rely on the `"main"` channel to be published on, as it would happen within the CherryPy's mainloop, you should find a way to publish to it within the other framework's mainloop.
- Start the CherryPy's engine. This will publish to the `"start"` channel of the bus.

  ```
  cherrypy.engine.start()
  ```

- Stop the CherryPy's engine when you terminate. This will publish to the `"stop"` channel of the bus.

  ```
  cherrypy.engine.stop()
  ```

- Do not call `cherrypy.engine.block()`.
- Disable the built-in HTTP server since it will not be used.

  ```
  cherrypy.server.unsubscribe()
  ```

- Disable autoreload. Usually other frameworks won't react well to it, or sometimes, provide the same feature.

  ```
  cherrypy.config.update({'engine.autoreload.on': False})
  ```

- Disable CherryPy signals handling. This may not be needed, it depends on how the other framework handles them.

  ```
  cherrypy.engine.signals.subscribe()
  ```

- Use the `"embedded"` environment configuration scheme.

  ```
  cherrypy.config.update({'environment': 'embedded'})
  ```

  Essentially this will disable the following:

- Stdout logging
- Autoreloader
- Configuration checker
- Headers logging on error
- Tracebacks in error
- Mismatched params error during dispatching
- Signals (SIGHUP, SIGTERM)

8.7.2 Tornado

You can use tornado HTTP server as follow:

```
import cherrypy

class Root(object):
    @cherrypy.expose
    def index(self):
        return "Hello World!"

if __name__ == '__main__':
    import tornado
    import tornado.httpserver
    import tornado.wsgi

    # our WSGI application
    wsgiapp = cherrypy.tree.mount(Root())

    # Disable the autoreload which won't play well
    cherrypy.config.update({'engine.autoreload.on': False})

    # let's not start the CherryPy HTTP server
    cherrypy.server.unsubscribe()

    # use CherryPy's signal handling
    cherrypy.engine.signals.subscribe()

    # Prevent CherryPy logs to be propagated
    # to the Tornado logger
    cherrypy.log.error_log.propagate = False

    # Run the engine but don't block on it
    cherrypy.engine.start()

    # Run thr tornado stack
    container = tornado.wsgi.WSGIContainer(wsgiapp)
    http_server = tornado.httpserver.HTTPServer(container)
    http_server.listen(8080)
    # Publish to the CherryPy engine as if
    # we were using its mainloop
    tornado.ioloop.PeriodicCallback(lambda: cherrypy.engine.publish('main'), 100).
↪start()
    tornado.ioloop.IOLoop.instance().start()
```

8.7.3 Twisted

You can use Twisted HTTP server as follow:

```
import cherrypy

from twisted.web.wsgi import WSGIResource
from twisted.internet import reactor
from twisted.internet import task

# Our CherryPy application
class Root(object):
    @cherrypy.expose
    def index(self):
        return "hello world"

# Create our WSGI app from the CherryPy application
wsgiapp = cherrypy.tree.mount(Root())

# Configure the CherryPy's app server
# Disable the autoreload which won't play well
cherrypy.config.update({'engine.autoreload.on': False})

# We will be using Twisted HTTP server so let's
# disable the CherryPy's HTTP server entirely
cherrypy.server.unsubscribe()

# If you'd rather use CherryPy's signal handler
# Uncomment the next line. I don't know how well this
# will play with Twisted however
#cherrypy.engine.signals.subscribe()

# Publish periodically onto the 'main' channel as the bus mainloop would do
task.LoopingCall(lambda: cherrypy.engine.publish('main')).start(0.1)

# Tie our app to Twisted
reactor.addSystemEventTrigger('after', 'startup', cherrypy.engine.start)
reactor.addSystemEventTrigger('before', 'shutdown', cherrypy.engine.exit)
resource = WSGIResource(reactor, reactor.getThreadPool(), wsgiapp)
```

Notice how we attach the bus methods to the Twisted's own lifecycle.

Save that code into a module named `cptw.py` and run it as follows:

```
$ twistd -n web --port 8080 --wsgi cptw.wsgiapp
```

8.7.4 uwsgi

You can use uwsgi HTTP server as follow:

```
import cherrypy

# Our CherryPy application
class Root(object):
    @cherrypy.expose
    def index(self):
```

(continues on next page)

(continued from previous page)

```
        return "hello world"

cherrypy.config.update({'engine.autoreload.on': False})
cherrypy.server.unsubscribe()
cherrypy.engine.start()

wsgiapp = cherrypy.tree.mount(Root())
```

Save this into a Python module called `mymod.py` and run it as follows:

```
$ uwsgi --socket 127.0.0.1:8080 --protocol=http --wsgi-file mymod.py --callable␣
↪wsgiapp
```

8.8 Virtual Hosting

CherryPy has support for virtual-hosting. It does so through a dispatchers that locate the appropriate resource based on the requested domain.

Below is a simple example for it:

```
import cherrypy

class Root(object):
    def __init__(self):
        self.app1 = App1()
        self.app2 = App2()

class App1(object):
    @cherrypy.expose
    def index(self):
        return "Hello world from app1"

class App2(object):
    @cherrypy.expose
    def index(self):
        return "Hello world from app2"

if __name__ == '__main__':
    hostmap = {
        'company.com:8080': '/app1',
        'home.net:8080': '/app2',
    }

    config = {
        'request.dispatch': cherrypy.dispatch.VirtualHost(**hostmap)
    }

    cherrypy.quickstart(Root(), '/', {'/': config})
```

In this example, we declare two domains and their ports:

- company.com:8080
- home.net:8080

Thanks to the `cherrypy.dispatch.VirtualHost` dispatcher, we tell CherryPy which application to dispatch to when a request arrives. The dispatcher looks up the requested domain and call the according application.

Note: To test this example, simply add the following rules to your `hosts` file:

```
127.0.0.1       company.com
127.0.0.1       home.net
```

8.9 Reverse-proxying

8.9.1 Apache

8.9.2 Nginx

nginx is a fast and modern HTTP server with a small footprint. It is a popular choice as a reverse proxy to application servers such as CherryPy.

This section will not cover the whole range of features nginx provides. Instead, it will simply provide you with a basic configuration that can be a good starting point.

```
upstream apps {
   server 127.0.0.1:8080;
   server 127.0.0.1:8081;
}

gzip_http_version 1.0;
gzip_proxied      any;
gzip_min_length   500;
gzip_disable      "MSIE [1-6]\.";
gzip_types        text/plain text/xml text/css
                  text/javascript
                  application/javascript;

server {
   listen 80;
   server_name  www.example.com;

   access_log  /app/logs/www.example.com.log combined;
   error_log  /app/logs/www.example.com.log;

   location ^~ /static/  {
      root /app/static/;
   }

   location / {
      proxy_pass         http://apps;
      proxy_redirect     off;
      proxy_set_header   Host $host;
      proxy_set_header   X-Real-IP $remote_addr;
      proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header   X-Forwarded-Host $server_name;
   }
}
```

Edit this configuration to match your own paths. Then, save this configuration into a file under `/etc/nginx/conf.d/` (assuming Ubuntu). The filename is irrelevant. Then run the following commands:

```
$ sudo service nginx stop
$ sudo service nginx start
```

Hopefully, this will be enough to forward requests hitting the nginx frontend to your CherryPy application. The `upstream` block defines the addresses of your CherryPy instances.

It shows that you can load-balance between two application servers. Refer to the nginx documentation to understand how this achieved.

```
upstream apps {
   server 127.0.0.1:8080;
   server 127.0.0.1:8081;
}
```

Later on, this block is used to define the reverse proxy section.

Now, let's see our application:

```
import cherrypy

class Root(object):
    @cherrypy.expose
    def index(self):
        return "hello world"

if __name__ == '__main__':
    cherrypy.config.update({
        'server.socket_port': 8080,
        'tools.proxy.on': True,
        'tools.proxy.base': 'http://www.example.com'
    })
    cherrypy.quickstart(Root())
```

If you run two instances of this code, one on each port defined in the nginx section, you will be able to reach both of them via the load-balancing done by nginx.

Notice how we define the proxy tool. It is not mandatory and used only so that the CherryPy request knows about the true client's address. Otherwise, it would know only about the nginx's own address. This is most visible in the logs.

The `base` attribute should match the `server_name` section of the nginx configuration.

CHAPTER

NINE

SUPPORT

You've read the documentation and you've brushed up on the basics of Python and web development, but you still could use some help. Users have several options.

9.1 I have a question

If you have a question and cannot find an answer for it in issues or the the documentation, please create an issue.

Questions and their answers have great value for the community, and a tip is to really put the effort in and write a good explanation, you will get better and quicker answers. Examples are strongly encouraged.

9.2 I have found a bug

If no one have already, create an issue. Be sure to provide ample information, remember that any help won't be better than your explanation.

Unless something is very obviously wrong, you are likely to be asked to provide a working example, displaying the erroneous behaviour.

Note: While this might feel troublesome, a tip is to always make a separate example that have the same dependencies as your project. It is great for troubleshooting those annoying problems where you don't know if the problem is at your end or the components. Also, you can then easily fork and provide as an example. You will get answers and resolutions way quicker. Also, many other open source projects require it.

9.3 I have a feature request

Good stuff! Please create an issue! Note: Features are more likely to be added the more users they seem to benefit.

9.4 I want to converse

The gitter page is good for when you want to discuss in real time or get pointed in the right direction.

CHAPTER

TEN

FOR ENTERPRISE

CherryPy is available as part of the Tidelift Subscription.

The CherryPy maintainers and the maintainers of thousands of other packages are working with Tidelift to deliver one enterprise subscription that covers all of the open source you use.

Learn more.

CHAPTER

ELEVEN

CONTRIBUTE

CherryPy is a community-maintained, open-source project hosted at Github. The project actively encourages aspiring and experienced users to dive in and add their best contribution to the project.

How can you contribute? Well, first search the docs and the project page to see if someone has already reported your issue.

11.1 StackOverflow

On StackOverflow, there are questions tagged with 'cherrypy'. Answer unanswered questions, add an improved answer, clarify an answer with a comment, or ask more meaningful questions there. Earn reputation and share experience.

11.2 Filing Bug Reports

If you find a bug, an issue where the product doesn't behave as you expect, you may file a bug report at the project page. Be sure to include what your expectation was, what happened instead, details about your system that might be relevant, and steps that someone else could take to replicate your finding. The more detailed and exact your description, the better one of the volunteers on the project may be able to help resolve your issue.

11.3 Fixing Bugs

CherryPy has a number of open, reported issues. Some of them are complicated and difficult, but others are more straightforward and shovel-ready. Feel free to find one that you think you can solve or introduce yourself and ask for guidance in our gitter channel.

As you work through the issue and commit changes to your clone of the repository, be sure to add issue references to your changes (like "Fixes #999" or "Ref #999") so your changes link to the issue and vice-versa.

11.4 Writing Pull Requests

To contribute, first read How to write the perfect pull request and file your contribution with the CherryPy Project page.

CHAPTER

TWELVE

TESTING

- To run the regression tests, first install tox:

```
pip install 'tox>=2.5'
```

  then run it

```
tox
```

- To run individual tests type:

```
tox -- -k test_foo
```

CHAPTER

THIRTEEN

GLOSSARY

application A CherryPy application is simply a class instance containing at least one page handler.

controller Loose name commonly given to a class owning at least one exposed method

exposed A Python function or method which has an attribute called *exposed* set to `True`. This attribute can be set directly or via the *cherrypy.expose()* decorator.

```
@cherrypy.expose
def method(...):
    ...
```

is equivalent to:

```
def method(...):
    ...
method.exposed = True
```

page handler Name commonly given to an exposed method

CHAPTER

FOURTEEN

HISTORY

14.1 v18.6.1

- #1849 via PR #1879: Fixed XLF flag in gzip header emitted by gzip compression tool per **RFC 1952#section-2.3.1** – by @webknjaz.
- #1874: Restricted depending on pywin32 only under CPython so that it won't get pulled-in under PyPy – by @webknjaz.

14.2 v18.6.0

17 Apr 2020

- #1776 via PR #1851: Add support for UTF-8 encoded attachment file names in `Content-Disposition` header via **RFC 6266#appendix-D**.

14.3 v18.5.0

27 Nov 2019

- #1827: Fixed issue where bytes values in a `HeaderMap` would be converted to strings.
- PR #1826: Rely on jaraco.collections for its case-insensitive dictionary support.

14.4 v18.4.0

03 Nov 2019

- PR #1715: Fixed issue in cpstats where the `data/` endpoint would fail with encoding errors on Python 3.
- PR #1821: Simplify the passthrough of parameters to `CPWebCase.getPage` to cheroot. CherryPy now requires cheroot 8.2.1 or later.

14.5 v18.3.0

02 Oct 2019

- PR #1806: Support handling multiple exceptions when processing hooks as reported in #1770.

14.6 v18.2.0

03 Sep 2019

- File-based sessions no longer attempt to remove the lock files when releasing locks, instead deferring to the default behavior of zc.lockfile. Fixes #1391 and #1779.
- PR #1794: Add native support for `308 Permanent Redirect` usable via `raise cherrypy.HTTPRedirect('/new_uri', 308)`.

14.7 v18.1.2

23 Jun 2019

- Fixed #1377 via PR #1785: Restore a native WSGI-less HTTP server support.
- PR #1769: Reduce log level for non-error events in win32.py

14.8 v18.1.1

27 Mar 2019

- PR #1774 reverts PR #1759 as new evidence emerged that the original behavior was intentional. Re-opens #1758.

14.9 v18.1.0

09 Dec 2018

- #1758 via PR #1759: In the bus, when awaiting a state change, only publish after the state has changed.

14.10 v18.0.1

09 Sep 2018

- #1738 via PR #1736: Restore support for 'bytes' in response headers.
- Substantial removal of Python 2 compatibility code.

14.11 v18.0.0

01 Sep 2018

- #1730: Drop support for Python 2.7. CherryPy 17 will remain an LTS release for bug and security fixes.
- Drop support for Python 3.4.

14.12 v17.4.2

23 Jun 2019

- Fixed #1377 by backporting PR #1785 via PR #1786: Restore a native WSGI-less HTTP server support.

14.13 v17.4.1

23 Nov 2018

- #1738 via PR #1755: Restore support for 'bytes' in response headers (backport from v18.0.1).

14.14 v17.4.0

19 Aug 2018

- a95e619f: When setting Response Body, reject Unicode values, making behavior on Python 2 same as on Python 3.
- Other inconsequential refactorings.

14.15 v17.3.0

16 Aug 2018

- #1193 via PR #1729: Rely on zc.lockfile for session concurrency support.

14.16 v17.2.0

14 Aug 2018

- #1690 via PR #1692: Prevent orphaned Event object in cached 304 response.

14.17 v17.1.0

14 Aug 2018

- #1694 via PR #1695: Add support for accepting uploaded files with non-ascii filenames per RFC 5987.

14.18 v17.0.0

10 Jul 2018

- #1673: CherryPy now allows namespace packages for its dependencies. Environments that cannot handle namespace packgaes like py2exe will need to add such support or pin to older CherryPy versions.

14.19 v16.0.3

10 Jul 2018

- #1722: Pinned the `tempora` dependency against version 1.13 to avoid pulling in namespace packages.

14.20 v16.0.2

18 Jun 2018

- #1716 via PR #1717: Fixed handling of url-encoded parameters in digest authentication handling, correcting regression in v14.2.0.
- #1719 via 1d41828: Digest-auth tool will now return a status code of 401 for when a scheme other than 'digest' is indicated.

14.21 v16.0.0

16 Jun 2018

- #1688 via 38ad1da: Removed `basic_auth` and `digest_auth` tools and the `httpauth` module, which have been officially deprecated earlier in v14.0.0.
- Removed deprecated properties:
  - `cherrypy._cpreqbody.Entity.type` deprecated in favor of *`cherrypy._cpreqbody.Entity.content_type`*
  - `cherrypy._cprequest.Request.body_params` deprecated in favor of `cherrypy._cprequest.RequestBody.params`
- #1377: In _cp_native server, set `req.status` using bytes (fixed in PR #1712).
- #1697 via 841f795: Fixed error on Python 3.7 with AutoReloader when `__file__` is `None`.
- #1713 via 15aa80d: Fix warning emitted during test run.
- #1370 via 38f199c: Fail with HTTP 400 for invalid headers.

14.22 v15.0.0

11 May 2018

- #1708: Removed components from webtest that were removed in the refactoring of cheroot.test.webtest for cheroot 6.1.0.

14.23 v14.2.0

22 Apr 2018

- #1680 via PR #1683: Basic Auth and Digest Auth tools now support **RFC 7617** UTF-8 charset decoding where possible, using latin-1 as a fallback.

14.24 v14.1.0

19 Apr 2018

- Cheroot PR #37: Add support for peercreds lookup over UNIX domain socket. This enables app to automatically identify "who's on the other end of the wire".

  This is how you enable it:

  ```
  server.peercreds: True
  server.peercreds_resolve: True
  ```

  The first option will put remote numeric data to WSGI env vars: app's PID, user's id and group.

  Second option will resolve that into user and group names.

  To prevent expensive syscalls, data is cached on per connection basis.

14.25 v14.0.1

22 Mar 2018

- #1700: Improve windows pywin32 dependency declaration via conditional extras.

14.26 v14.0.0

04 Feb 2018

- #1688: Officially deprecated `basic_auth` and `digest_auth` tools and the `httpauth` module, triggering DeprecationWarnings if they're used. Applications should instead adapt to use the more recent `auth_basic` and `auth_digest` tools. This deprecated functionality will be removed in a subsequent release soon.
- Removed `DeprecatedTool` and the long-deprecated and disabled `tidy` and `nsgmls` tools. See the rationale for this change.

14.27 v13.1.0

17 Dec 2017

- #1231 via PR #1654: CaseInsensitiveDict now re-uses the generalized functionality from `jaraco.collections` to provide a more complete interface for a CaseInsensitiveDict and HeaderMap.

  Users are encouraged to use the implementation from jaraco.collections except when dealing with headers in CherryPy.

14.28 v13.0.1

17 Dec 2017

- PR #1671: Restore support for installing CherryPy into environments hostile to namespace packages, broken since the 11.1.0 release.

14.29 v13.0.0

04 Dec 2017

- #1666: Drop support for Python 3.3.

14.30 v12.0.2

03 Dec 2017

- #1665: In request processing, when an invalid cookie is received, render the actual error message reported rather than guessing (sometimes incorrectly) what error occurred.

14.31 v12.0.1

20 Nov 2017

- Fixed issues importing *`cherrypy.test.webtest`* (by creating a module and importing classes from `cheroot`) and added a corresponding `DeprecationWarning`.

14.32 v12.0.0

17 Nov 2017

- Drop support for Python 3.1 and 3.2.
- #1625: Removed response timeout and timeout monitor and related exceptions, as it not possible to interrupt a request. Servers that wish to exit a request prematurely are recommended to monitor `response.time` and raise an exception or otherwise act accordingly.

  Servers that previously disabled timeouts by invoking `cherrypy.engine.timeout_monitor.unsubscribe()` will now crash. For forward-compatibility with this release on older versions of CherryPy, disable timeouts using the config option:

```
'engine.timeout_monitor.on': False,
```

Or test for the presence of the timeout_monitor attribute:

```
with contextlib2.suppress(AttributeError):
    cherrypy.engine.timeout_monitor.unsubscribe()
```

Additionally, the `TimeoutError` exception has been removed, as it's no longer called anywhere. If your application benefits from this Exception, please comment in the linked ticket describing the use case, and we'll help devise a solution or bring the exception back.

14.33 v11.3.0

- Bump to cheroot 5.9.0.
- `cherrypy.test.webtest` module is now merged with the `cheroot.test.webtest` module. The CherryPy name is retained for now for compatibility and will be removed eventually.

14.34 v11.2.0

13 Nov 2017

- `cherrypy.engine.subscribe` now may be called without a callback, in which case it returns a decorator expecting the callback.
- PR #1656: Images are now compressed using lossless compression and consume less space.

14.35 v11.1.0

28 Oct 2017

- PR #1611: Expose default status logic for a redirect as `HTTPRedirect.default_status`.
- PR #1615: `HTTPRedirect.status` is now an instance property and derived from the value in `args`. Although it was previously possible to set the property on an instance, and this change prevents that possibilty, CherryPy never relied on that behavior and we presume no applications depend on that interface.
- #1627: Fixed issue in proxy tool where more than one port would appear in the `request.base` and thus in `cherrypy.url`.
- PR #1645: Added new log format markers:
  - `i` holds a per-request UUID4
  - `z` outputs UTC time in format of RFC 3339
  - `cherrypy._cprequest.Request.unique_id.uuid4` now has lazily invocable UUID4
- #1646: Improve http status conversion helper.
- PR #1638: Always use backslash for path separator when processing paths in staticdir.
- #1190: Fix gzip, caching, and staticdir tools integration. Makes cache of gzipped content valid.
- Requires cheroot 5.8.3 or later.

- Also, many improvements around continuous integration and code quality checks.

This release contained an unintentional regression in environments that are hostile to namespace packages, such as Pex, Celery, and py2exe. See PR #1671 for details.

14.36 v11.0.0

08 Jul 2017

- #1607: Dropped support for Python 2.6.

14.37 v10.2.2

17 May 2017

- #1595: Fixed over-eager normalization of paths in cherrypy.url.

14.38 v10.2.1

13 Mar 2017

- Remove unintended dependency on `graphviz` in Python 2.6.

14.39 v10.2.0

12 Mar 2017

- PR #1580: `CPWSGIServer.version` now reported as `CherryPy/x.y.z Cheroot/x.y.z`. Bump to cheroot 5.2.0.
- The codebase is now **PEP 8** complaint, flake8 linter is enabled in TravisCI by default.
- Max line restriction is now set to 120 for flake8 linter.
- **PEP 257** linter runs as separate allowed failure job in Travis CI.
- A few bugs related to undeclared variables have been fixed.
- `pre-commit` testing goes faster due to enabled caching.

14.40 v10.1.1

18 Feb 2017

- #1342: Fix AssertionError on shutdown.

14.41 v10.1.0

07 Feb 2017

- Bump to cheroot 5.1.0.
- #794: Prefer setting max-age for session cookie expiration, moving MSIE hack into a function documenting its purpose.

14.42 v10.0.0

20 Jan 2017

- #1332: CherryPy now uses portend for checking and waiting on ports for startup and teardown checks. The following names are no longer present:
  - cherrypy._cpserver.client_host
  - cherrypy._cpserver.check_port
  - cherrypy._cpserver.wait_for_free_port
  - cherrypy._cpserver.wait_for_occupied_port
  - cherrypy.process.servers.check_port
  - cherrypy.process.servers.wait_for_free_port
  - cherrypy.process.servers.wait_for_occupied_port

  Use this functionality from the portend package directly.

14.43 v9.0.0

19 Jan 2017

- #1481: Move functionality from cherrypy.wsgiserver to the cheroot 5.0 project.

14.44 v8.9.1

16 Jan 2017

- #1537: Restore dependency on pywin32 for Python 3.6.

14.45 v8.9.0

13 Jan 2017

- PR #1547: Replaced `cherryd` distutils script with a setuptools console entry point.

  When running CherryPy in daemon mode, the forked process no longer changes directory to `/`. If that behavior is something on which your application relied and should rely, please file a ticket with the project.

14.46 v8.8.0

09 Jan 2017

- PR #1528: Allow a timeout of 0 to server.

14.47 v8.7.0

31 Dec 2016

- #645: Setting a bind port of 0 will bind to an ephemeral port.

14.48 v8.6.0

27 Dec 2016

- #1538 and #1090: Removed cruft from the setup script and instead rely on include_package_data to ensure the relevant files are included in the package. Note, this change does cause LICENSE.md no longer to be included in the installed package.

14.49 v8.5.0

26 Dec 2016

- The pyOpenSSL support is now included on Python 3 builds, removing the last disparity between Python 2 and Python 3 in the CherryPy package. This change is one small step in consideration of #1399. This change also fixes RPM builds, as reported in #1149.

14.50 v8.4.0

26 Dec 2016

- #1532: Also release wheels for Python 2, enabling offline installation.

14.51 v8.3.1

25 Dec 2016

- #1537: Disable dependency on pypiwin32 on Python 3.6 until a viable build of pypiwin32 can be made on that Python version.

14.52 v8.3.0

24 Dec 2016

- Consolidated some documentation and include the more concise readme in the package long description, as found on PyPI.

14.53 v8.2.0

23 Dec 2016

- #1463: CherryPy tests are now run under pytest and invoked using tox.

14.54 v8.1.3

16 Dec 2016

- #1530: Fix the issue with TypeError being swallowed by decorated handlers.

14.55 v8.1.2

28 Sep 2016

- #1508

14.56 v8.1.1

27 Sep 2016

- #1497: Handle errors thrown by `ssl_module: 'builtin'` when client opens connection to HTTPS port using HTTP.
- #1350: Fix regression introduced in v6.1.0 where environment construction for WSGIGateway_u0 was passing one parameter and not two.
- Other miscellaneous fixes.

14.57 v8.1.0

04 Sep 2016

- #1473: `HTTPError` now also works as a context manager.
- #1487: The sessions tool now accepts a `storage_class` parameter, which supersedes the new deprecated `storage_type` parameter. The `storage_class` should be the actual Session subclass to be used.
- Releases now use `setuptools_scm` to track the release versions. Therefore, releases can be cut by simply tagging a commit in the repo. Versions numbers are now stored in exactly one place.

14.58 v8.0.1

03 Sep 2016

- #1489 via PR #1493: Additionally reject anything else that's not bytes.
- #1492: systemd socket activation.

14.59 v8.0.0

02 Sep 2016

- #1483: Remove Deprecated constructs:
  - `cherrypy.lib.http` module.
  - `unrepr`, `modules`, and `attributes` in `cherrypy.lib`.
- PR #1476: Drop support for python-memcached<1.58
- #1401: Handle NoSSLErrors.
- #1489: In `wsgiserver.WSGIGateway.respond`, the application must now yield bytes and not text, as the spec requires. If text is received, it will now raise a ValueError instead of silently encoding using ISO-8859-1.
- Removed unicode filename from the package, working around pypa/pip#3894 and pypa/setuptools#704.

14.60 v7.1.0

25 Jul 2016

- PR #1458: Implement systemd's socket activation mechanism for CherryPy servers, based on work sponsored by Endless Computers.

  Socket Activation allows one to setup a system so that systemd will sit on a port and start services 'on demand' (a little bit like inetd and xinetd used to do).

14.61 v7.0.0

24 Jul 2016

Removed the long-deprecated backward compatibility for legacy config keys in the engine. Use the config for the namespaced-plugins instead:

- autoreload_on -> autoreload.on
- autoreload_frequency -> autoreload.frequency
- autoreload_match -> autoreload.match
- reload_files -> autoreload.files
- deadlock_poll_frequency -> timeout_monitor.frequency

14.62 v6.2.1

24 Jul 2016

- #1460: Fix KeyError in Bus.publish when signal handlers set in config.

14.63 v6.2.0

18 Jul 2016

- #1441: Added tool to automatically convert request params based on type annotations (primarily in Python 3). For example:

```
@cherrypy.tools.params()
def resource(self, limit: int):
    assert isinstance(limit, int)
```

14.64 v6.1.1

16 Jul 2016

- Issue #1411: Fix issue where autoreload fails when the host interpreter for CherryPy was launched using `python -m`.

14.65 v6.1.0

14 Jul 2016

- Combined wsgiserver2 and wsgiserver3 modules into a single module, `cherrypy.wsgiserver`.

14.66 v6.0.2

23 Jun 2016

- Issue PR #1445: Correct additional typos.

14.67 v6.0.1

06 Jun 2016

- Issue #1444: Correct typos in `@cherrypy.expose` decorators.

14.68 v6.0.0

05 Jun 2016

- Setuptools is now required to build CherryPy. Pure distutils installs are no longer supported. This change allows CherryPy to depend on other packages and re-use code from them. It's still possible to install pre-built CherryPy packages (wheels) using pip without Setuptools.
- six is now a requirement and subsequent requirements will be declared in the project metadata.
- #1440: Back out changes from PR #1432 attempting to fix redirects with Unicode URLs, as it also had the unintended consequence of causing the 'Location' to be `bytes` on Python 3.
- `cherrypy.expose` now works on classes.
- `cherrypy.config` decorator is now used throughout the code internally.

14.69 v5.6.0

05 Jun 2016

- `@cherrypy.expose` now will also set the exposed attribute on a class.
- Rewrote all tutorials and internal usage to prefer the decorator usage of `expose` rather than setting the attribute explicitly.
- Removed test-specific code from tutorials.

14.70 v5.5.0

05 Jun 2016

- #1397: Fix for filenames with semicolons and quote characters in filenames found in headers.
- #1311: Added decorator for registering tools.
- #1194: Use simpler encoding rules for SCRIPT_NAME and PATH_INFO environment variables in CherryPy Tree allowing non-latin characters to pass even when `wsgi.version` is not `u.0`.
- #1352: Ensure that multipart fields are decoded even when cached in a file.

14.71 v5.4.0

10 May 2016

- `cherrypy.test.webtest.WebCase` now honors a 'WEBTEST_INTERACTIVE' environment variable to disable interactive tests (still enabled by default). Set to '0' or 'false' or 'False' to disable interactive tests.
- #1408: Fix AttributeError when listiterator was accessed using the `next` attribute.
- #748: Removed `cherrypy.lib.sessions.PostgresqlSession`.
- PR #1432: Fix errors with redirects to Unicode URLs.

14.72 v5.3.0

30 Apr 2016

- #1202: Add support for specifying a certificate authority when serving SSL using the built-in SSL support.
- Use ssl.create_default_context when available.
- #1392: Catch platform-specific socket errors on OS X.
- #1386: Fix parsing of URIs containing `://` in the path part.

14.73 v5.2.0

30 Apr 2016

- #1410: Moved hosting to Github (cherrypy/cherrypy).

14.74 v5.1.0

- Bugfix issue #1315 for `test_HTTP11_pipelining` test in Python 3.5
- Bugfix issue #1382 regarding the keyword arguments support for Python 3 on the config file.
- Bugfix issue #1406 for `test_2_KeyboardInterrupt` test in Python 3.5. by monkey patching the HTTPRequest given a bug on CPython that is affecting the testsuite (https://bugs.python.org/issue23377).
- Add additional parameter `raise_subcls` to the tests helpers `openURL` and `CPWebCase.getPage` to have finer control on which exceptions can be raised.
- Add support for direct keywords on the calls (e.g. `foo=bar`) on the config file under Python 3.
- Add additional validation to determine if the process is running as a daemon on `cherrypy.process.plugins.SignalHandler` to allow the execution of the testsuite under CI tools.

14.75 v5.0.1

- Bugfix for NameError following #94.

14.76 v5.0.0

- Removed deprecated support for `ssl_certificate` and `ssl_private_key` attributes and implicit construction of SSL adapter on Python 2 WSGI servers.
- Default SSL Adapter on Python 2 is the builtin SSL adapter, matching Python 3 behavior.
- Pull request #94: In proxy tool, defer to Host header for resolving the base if no base is supplied.

14.77 v4.0.0

- Drop support for Python 2.5 and earlier.
- No longer build Windows installers by default.

14.78 v3.8.2

- Pull Request #116: Correct InternalServerError when null bytes in static file path. Now responds with 404 instead.

14.79 v3.8.0

- Pull Request #96: Pass `exc_info` to logger as keyword rather than formatting the error and injecting into the message.

14.80 v3.7.0

- CherryPy daemon may now be invoked with `python -m cherrypy` in addition to the `cherryd` script.
- Issue #1298: Fix SSL handling on CPython 2.7 with builtin SSL module and pyOpenSSL 0.14. This change will break PyPy for now.
- Several documentation fixes.

14.81 v3.6.0

- Fixed HTTP range headers for negative length larger than content size.
- Disabled universal wheel generation as wsgiserver has Python duality.
- Pull Request #42: Correct TypeError in `check_auth` when encrypt is used.
- Pull Request #59: Correct signature of HandlerWrapperTool.
- Pull Request #60: Fix error in SessionAuth where login_screen was incorrectly used.
- Issue #1077: Support keyword-only arguments in dispatchers (Python 3).
- Issue #1019: Allow logging host name in the access log.
- Pull Request #50: Fixed race condition in session cleanup.

14.82 v3.5.0

- Issue #1301: When the incoming queue is full, now reject additional connections. This functionality was added to CherryPy 3.0, but unintentionally lost in 3.1.

14.83 v3.4.0

- Miscellaneous quality improvements.

14.84 v3.3.0

CherryPy adopts semver.

CHAPTER

FIFTEEN

CHERRYPY

15.1 cherrypy package

15.1.1 Subpackages

cherrypy.lib package

Submodules

cherrypy.lib.auth_basic module

HTTP Basic Authentication tool.

This module provides a CherryPy 3.x tool which implements the server-side of HTTP Basic Access Authentication, as described in **RFC 2617**.

Example usage, using the built-in checkpassword_dict function which uses a dict as the credentials store:

```
userpassdict = {'bird' : 'bebop', 'ornette' : 'wayout'}
checkpassword = cherrypy.lib.auth_basic.checkpassword_dict(userpassdict)
basic_auth = {'tools.auth_basic.on': True,
              'tools.auth_basic.realm': 'earth',
              'tools.auth_basic.checkpassword': checkpassword,
              'tools.auth_basic.accept_charset': 'UTF-8',
}
app_config = { '/' : basic_auth }
```

cherrypy.lib.auth_basic.**_try_decode**(*subject*, *charsets*)

cherrypy.lib.auth_basic.**basic_auth**(*realm*, *checkpassword*, *debug=False*, *accept_charset='utf-8'*)

A CherryPy tool which hooks at before_handler to perform HTTP Basic Access Authentication, as specified in **RFC 2617** and **RFC 7617**.

If the request has an 'authorization' header with a 'Basic' scheme, this tool attempts to authenticate the credentials supplied in that header. If the request has no 'authorization' header, or if it does but the scheme is not 'Basic', or if authentication fails, the tool sends a 401 response with a 'WWW-Authenticate' Basic header.

realm A string containing the authentication realm.

checkpassword A callable which checks the authentication credentials. Its signature is checkpassword(realm, username, password). where username and password are the values obtained from the request's 'authorization' header. If authentication succeeds, checkpassword returns True, else it returns False.

cherrypy.lib.auth_basic.**checkpassword_dict**(*user_password_dict*)
Returns a checkpassword function which checks credentials against a dictionary of the form: {username : password}.

If you want a simple dictionary-based authentication scheme, use checkpassword_dict(my_credentials_dict) as the value for the checkpassword argument to basic_auth().

cherrypy.lib.auth_digest module

HTTP Digest Authentication tool.

An implementation of the server-side of HTTP Digest Access Authentication, which is described in RFC 2617.

Example usage, using the built-in get_ha1_dict_plain function which uses a dict of plaintext passwords as the credentials store:

```
userpassdict = {'alice' : '4x5istwelve'}
get_ha1 = cherrypy.lib.auth_digest.get_ha1_dict_plain(userpassdict)
digest_auth = {'tools.auth_digest.on': True,
               'tools.auth_digest.realm': 'wonderland',
               'tools.auth_digest.get_ha1': get_ha1,
               'tools.auth_digest.key': 'a565c27146791cfb',
               'tools.auth_digest.accept_charset': 'UTF-8',
}
app_config = { '/' : digest_auth }
```

cherrypy.lib.auth_digest.**H**(*s*)
The hash function H

class cherrypy.lib.auth_digest.**HttpDigestAuthorization**(*auth_header*, *http_method*, *debug=False*, *accept_charset='UTF-8'*)
Bases: object

Parses a Digest Authorization header and performs re-calculation of the digest.

HA2(*entity_body=''*)
Returns the H(A2) string. See RFC 2617 section 3.2.2.3.

errmsg(*s*)

is_nonce_stale(*max_age_seconds=600*)
Returns True if a validated nonce is stale. The nonce contains a timestamp in plaintext and also a secure hash of the timestamp. You should first validate the nonce to ensure the plaintext timestamp is not spoofed.

classmethod matches(*header*)

request_digest(*ha1*, *entity_body=''*)
Calculates the Request-Digest. See RFC 2617 section 3.2.2.1.

ha1 The HA1 string obtained from the credentials store.

entity_body If 'qop' is set to 'auth-int', then A2 includes a hash of the "entity body". The entity body is the part of the message which follows the HTTP headers. See RFC 2617 section 4.3. This refers to the entity the user agent sent in the request which has the Authorization header. Typically GET requests don't have an entity, and POST requests do.

scheme = 'digest'

validate_nonce(*s*, *key*)
Validate the nonce. Returns True if nonce was generated by synthesize_nonce() and the timestamp is not spoofed, else returns False.

s A string related to the resource, such as the hostname of the server.

key A secret string known only to the server.

Both s and key must be the same values which were used to synthesize the nonce we are trying to validate.

cherrypy.lib.auth_digest.**TRACE**(*msg*)

cherrypy.lib.auth_digest.**_get_charset_declaration**(*charset*)

cherrypy.lib.auth_digest.**_respond_401**(*realm*, *key*, *accept_charset*, *debug*, ***kwargs*)
Respond with 401 status and a WWW-Authenticate header

cherrypy.lib.auth_digest.**_try_decode_header**(*header*, *charset*)

cherrypy.lib.auth_digest.**digest_auth**(*realm*, *get_ha1*, *key*, *debug=False*, *accept_charset='utf-8'*)
A CherryPy tool that hooks at before_handler to perform HTTP Digest Access Authentication, as specified in RFC 2617.

If the request has an 'authorization' header with a 'Digest' scheme, this tool authenticates the credentials supplied in that header. If the request has no 'authorization' header, or if it does but the scheme is not "Digest", or if authentication fails, the tool sends a 401 response with a 'WWW-Authenticate' Digest header.

realm A string containing the authentication realm.

get_ha1 A callable that looks up a username in a credentials store and returns the HA1 string, which is defined in the RFC to be MD5(username : realm : password). The function's signature is: `get_ha1(realm, username)` where username is obtained from the request's 'authorization' header. If username is not found in the credentials store, get_ha1() returns None.

key A secret string known only to the server, used in the synthesis of nonces.

cherrypy.lib.auth_digest.**get_ha1_dict**(*user_ha1_dict*)
Returns a get_ha1 function which obtains a HA1 password hash from a dictionary of the form: {username : HA1}.

If you want a dictionary-based authentication scheme, but with pre-computed HA1 hashes instead of plain-text passwords, use get_ha1_dict(my_userha1_dict) as the value for the get_ha1 argument to digest_auth().

cherrypy.lib.auth_digest.**get_ha1_dict_plain**(*user_password_dict*)
Returns a get_ha1 function which obtains a plaintext password from a dictionary of the form: {username : password}.

If you want a simple dictionary-based authentication scheme, with plaintext passwords, use get_ha1_dict_plain(my_userpass_dict) as the value for the get_ha1 argument to digest_auth().

cherrypy.lib.auth_digest.**get_ha1_file_htdigest**(*filename*)
Returns a get_ha1 function which obtains a HA1 password hash from a flat file with lines of the same format as that produced by the Apache htdigest utility. For example, for realm 'wonderland', username 'alice', and password '4x5istwelve', the htdigest line would be:

```
alice:wonderland:3238cdfe91a8b2ed8e39646921a02d4c
```

If you want to use an Apache htdigest file as the credentials store, then use get_ha1_file_htdigest(my_htdigest_file) as the value for the get_ha1 argument to digest_auth(). It is recommended that the filename argument be an absolute path, to avoid problems.

cherrypy.lib.auth_digest.**md5_hex**(*s*)

cherrypy.lib.auth_digest.**synthesize_nonce**(*s*, *key*, *timestamp=None*)
Synthesize a nonce value which resists spoofing and can be checked for staleness. Returns a string suitable as the value for 'nonce' in the www-authenticate header.

s A string related to the resource, such as the hostname of the server.

key A secret string known only to the server.

timestamp An integer seconds-since-the-epoch timestamp

cherrypy.lib.auth_digest.**www_authenticate**(*realm*, *key*, *algorithm='MD5'*, *nonce=None*, *qop='auth'*, *stale=False*, *accept_charset='UTF-8'*)
Constructs a WWW-Authenticate header for Digest authentication.

cherrypy.lib.caching module

CherryPy implements a simple caching system as a pluggable Tool. This tool tries to be an (in-process) HTTP/1.1-compliant cache. It's not quite there yet, but it's probably good enough for most sites.

In general, GET responses are cached (along with selecting headers) and, if another request arrives for the same resource, the caching Tool will return 304 Not Modified if possible, or serve the cached response otherwise. It also sets request.cached to True if serving a cached representation, and sets request.cacheable to False (so it doesn't get cached again).

If POST, PUT, or DELETE requests are made for a cached resource, they invalidate (delete) any cached response.

Usage

Configuration file example:

```
[/]
tools.caching.on = True
tools.caching.delay = 3600
```

You may use a class other than the default *MemoryCache* by supplying the config entry `cache_class`; supply the full dotted name of the replacement class as the config value. It must implement the basic methods `get`, `put`, `delete`, and `clear`.

You may set any attribute, including overriding methods, on the cache instance by providing them in config. The above sets the *delay* attribute, for example.

class cherrypy.lib.caching.**AntiStampedeCache**
Bases: dict

A storage system for cached items which reduces stampede collisions.

wait(*key*, *timeout=5*, *debug=False*)
Return the cached value for the given key, or None.

If timeout is not None, and the value is already being calculated by another thread, wait until the given timeout has elapsed. If the value is available before the timeout expires, it is returned. If not, None is returned, and a sentinel placed in the cache to signal other threads to wait.

If timeout is None, no waiting is performed nor sentinels used.

class cherrypy.lib.caching.**Cache**
Bases: object

Base class for Cache implementations.

clear()
Reset the cache to its initial, empty state.

delete()
Remove ALL cached variants of the current resource.

get()
Return the current variant if in the cache, else None.

put(*obj*, *size*)
Store the current variant in the cache.

class cherrypy.lib.caching.**MemoryCache**
Bases: *cherrypy.lib.caching.Cache*

An in-memory cache for varying response content.

Each key in self.store is a URI, and each value is an AntiStampedeCache. The response for any given URI may vary based on the values of "selecting request headers"; that is, those named in the Vary response header. We assume the list of header names to be constant for each URI throughout the lifetime of the application, and store that list in `self.store[uri].selecting_headers`.

The items contained in `self.store[uri]` have keys which are tuples of request header values (in the same order as the names in its selecting_headers), and values which are the actual responses.

antistampede_timeout = 5
Seconds to wait for other threads to release a cache lock.

clear()
Reset the cache to its initial, empty state.

debug = False

delay = 600
Seconds until the cached content expires; defaults to 600 (10 minutes).

delete()
Remove ALL cached variants of the current resource.

expire_cache()
Continuously examine cached objects, expiring stale ones.

This function is designed to be run in its own daemon thread, referenced at `self.expiration_thread`.

expire_freq = 0.1
Seconds to sleep between cache expiration sweeps.

get()
Return the current variant if in the cache, else None.

maxobj_size = 100000
The maximum size of each cached object in bytes; defaults to 100 KB.

maxobjects = 1000
The maximum number of cached objects; defaults to 1000.

maxsize = 10000000
The maximum size of the entire cache in bytes; defaults to 10 MB.

put(*variant*, *size*)
Store the current variant in the cache.

cherrypy.lib.caching.**expires**(*secs=0*, *force=False*, *debug=False*)
Tool for influencing cache mechanisms using the 'Expires' header.

secs Must be either an int or a datetime.timedelta, and indicates the number of seconds between response.time and when the response should expire. The 'Expires' header will be set to response.time + secs. If secs is zero, the 'Expires' header is set one year in the past, and the following "cache prevention" headers are also set:

- Pragma: no-cache
- Cache-Control': no-cache, must-revalidate

force If False, the following headers are checked:

- Etag
- Last-Modified
- Age
- Expires

If any are already present, none of the above response headers are set.

cherrypy.lib.caching.**get**(*invalid_methods=('POST', 'PUT', 'DELETE')*, *debug=False*, ***kwargs*)
Try to obtain cached output. If fresh enough, raise HTTPError(304).

If POST, PUT, or DELETE:

- invalidates (deletes) any cached response for this resource
- sets request.cached = False
- sets request.cacheable = False

else if a cached copy exists:

- sets request.cached = True
- sets request.cacheable = False
- sets response.headers to the cached values
- checks the cached Last-Modified response header against the current If-(Un)Modified-Since request headers; raises 304 if necessary.
- sets response.status and response.body to the cached values
- returns True

otherwise:

- sets request.cached = False
- sets request.cacheable = True
- returns False

cherrypy.lib.caching.**tee_output**()
Tee response output to cache storage. Internal.

cherrypy.lib.covercp module

Code-coverage tools for CherryPy.

To use this module, or the coverage tools in the test suite, you need to download 'coverage.py', either Gareth Rees' original implementation or Ned Batchelder's enhanced version:

To turn on coverage tracing, use the following code:

```
cherrypy.engine.subscribe('start', covercp.start)
```

DO NOT subscribe anything on the 'start_thread' channel, as previously recommended. Calling start once in the main thread should be sufficient to start coverage on all threads. Calling start again in each thread effectively clears any coverage data gathered up to that point.

Run your code, then use the `covercp.serve()` function to browse the results in a web browser. If you run this module from the command line, it will call `serve()` for you.

class cherrypy.lib.covercp.**CoverStats**(*coverage*, *root=None*)
Bases: object

annotated_file(*filename*, *statements*, *excluded*, *missing*)

index()

menu(*base='/'*, *pct='50'*, *showpct=''*, *exclude='python\\d\\.\\d|test|tut\\d|tutorial'*)

report(*name*)

cherrypy.lib.covercp.**_graft**(*path*, *tree*)

cherrypy.lib.covercp.**_percent**(*statements*, *missing*)

cherrypy.lib.covercp.**_show_branch**(*root*, *base*, *path*, *pct=0*, *showpct=False*, *exclude=''*, *coverage=<coverage.control.Coverage object>*)

cherrypy.lib.covercp.**_skip_file**(*path*, *exclude*)

cherrypy.lib.covercp.**get_tree**(*base*, *exclude*, *coverage=<coverage.control.Coverage object>*)
Return covered module names as a nested dict.

cherrypy.lib.covercp.**serve**(*path='/home/docs/checkouts/readthedocs.org/user_builds/cherrypy/envs/latest/lib/python3.6/site-packages/cherrypy/lib/coverage.cache'*, *port=8080*, *root=None*)

cherrypy.lib.covercp.**start**()

cherrypy.lib.cpstats module

CPStats, a package for collecting and reporting on program statistics.

Overview

Statistics about program operation are an invaluable monitoring and debugging tool. Unfortunately, the gathering and reporting of these critical values is usually ad-hoc. This package aims to add a centralized place for gathering statistical performance data, a structure for recording that data which provides for extrapolation of that data into more useful information, and a method of serving that data to both human investigators and monitoring software. Let's examine each of those in more detail.

Data Gathering

Just as Python's `logging` module provides a common importable for gathering and sending messages, performance statistics would benefit from a similar common mechanism, and one that does *not* require each package which wishes to collect stats to import a third-party module. Therefore, we choose to re-use the `logging` module by adding a *`statistics`* object to it.

That `logging.statistics` object is a nested dict. It is not a custom class, because that would:

1. require libraries and applications to import a third-party module in order to participate
2. inhibit innovation in extrapolation approaches and in reporting tools, and
3. be slow.

There are, however, some specifications regarding the structure of the dict.:

```
{
  +----"SQLAlchemy": {
  |         "Inserts": 4389745,
  |         "Inserts per Second":
  |             lambda s: s["Inserts"] / (time() - s["Start"]),
  |   C +---"Table Statistics": {
  |   o |         "widgets": {-----------+
N |   l |             "Rows": 1.3M,      | Record
a |   l |             "Inserts": 400,    |
m |   e |         },---------------------+
e |   c |         "froobles": {
s |   t |             "Rows": 7845,
p |   i |             "Inserts": 0,
a |   o |         },
c |   n +---},
e |         "Slow Queries":
  |             [{"Query": "SELECT * FROM widgets;",
  |               "Processing Time": 47.840923343,
  |               },
  |             ],
  +----},
}
```

The `logging.statistics` dict has four levels. The topmost level is nothing more than a set of names to introduce modularity, usually along the lines of package names. If the SQLAlchemy project wanted to participate, for example, it might populate the item `logging.statistics['SQLAlchemy']`, whose value would be a second-layer dict we call a "namespace". Namespaces help multiple packages to avoid collisions over key names, and make reports easier to read, to boot. The maintainers of SQLAlchemy should feel free to use more than one namespace if needed (such as 'SQLAlchemy ORM'). Note that there are no case or other syntax constraints on the namespace names; they should be chosen to be maximally readable by humans (neither too short nor too long).

Each namespace, then, is a dict of named statistical values, such as 'Requests/sec' or 'Uptime'. You should choose names which will look good on a report: spaces and capitalization are just fine.

In addition to scalars, values in a namespace MAY be a (third-layer) dict, or a list, called a "collection". For example, the CherryPy *`StatsTool`* keeps track of what each request is doing (or has most recently done) in a 'Requests' collection, where each key is a thread ID; each value in the subdict MUST be a fourth dict (whew!) of statistical data about each thread. We call each subdict in the collection a "record". Similarly, the *`StatsTool`* also keeps a list of slow queries, where each record contains data about each slow query, in order.

Values in a namespace or record may also be functions, which brings us to:

Extrapolation

The collection of statistical data needs to be fast, as close to unnoticeable as possible to the host program. That requires us to minimize I/O, for example, but in Python it also means we need to minimize function calls. So when you are designing your namespace and record values, try to insert the most basic scalar values you already have on hand.

When it comes time to report on the gathered data, however, we usually have much more freedom in what we can calculate. Therefore, whenever reporting tools (like the provided *StatsPage* CherryPy class) fetch the contents of `logging.statistics` for reporting, they first call *extrapolate_statistics* (passing the whole *statistics* dict as the only argument). This makes a deep copy of the statistics dict so that the reporting tool can both iterate over it and even change it without harming the original. But it also expands any functions in the dict by calling them. For example, you might have a 'Current Time' entry in the namespace with the value "lambda scope: time.time()". The "scope" parameter is the current namespace dict (or record, if we're currently expanding one of those instead), allowing you access to existing static entries. If you're truly evil, you can even modify more than one entry at a time.

However, don't try to calculate an entry and then use its value in further extrapolations; the order in which the functions are called is not guaranteed. This can lead to a certain amount of duplicated work (or a redesign of your schema), but that's better than complicating the spec.

After the whole thing has been extrapolated, it's time for:

Reporting

The *StatsPage* class grabs the `logging.statistics` dict, extrapolates it all, and then transforms it to HTML for easy viewing. Each namespace gets its own header and attribute table, plus an extra table for each collection. This is NOT part of the statistics specification; other tools can format how they like.

You can control which columns are output and how they are formatted by updating StatsPage.formatting, which is a dict that mirrors the keys and nesting of `logging.statistics`. The difference is that, instead of data values, it has formatting values. Use None for a given key to indicate to the StatsPage that a given column should not be output. Use a string with formatting (such as '%.3f') to interpolate the value(s), or use a callable (such as lambda v: v.isoformat()) for more advanced formatting. Any entry which is not mentioned in the formatting dict is output unchanged.

Monitoring

Although the HTML output takes pains to assign unique id's to each <td> with statistical data, you're probably better off fetching /cpstats/data, which outputs the whole (extrapolated) `logging.statistics` dict in JSON format. That is probably easier to parse, and doesn't have any formatting controls, so you get the "original" data in a consistently-serialized format. Note: there's no treatment yet for datetime objects. Try time.time() instead for now if you can. Nagios will probably thank you.

Turning Collection Off

It is recommended each namespace have an "Enabled" item which, if False, stops collection (but not reporting) of statistical data. Applications SHOULD provide controls to pause and resume collection by setting these entries to False or True, if present.

Usage

To collect statistics on CherryPy applications:

```
from cherrypy.lib import cpstats
appconfig['/']['tools.cpstats.on'] = True
```

To collect statistics on your own code:

```
import logging
# Initialize the repository
if not hasattr(logging, 'statistics'): logging.statistics = {}
# Initialize my namespace
mystats = logging.statistics.setdefault('My Stuff', {})
# Initialize my namespace's scalars and collections
mystats.update({
    'Enabled': True,
    'Start Time': time.time(),
    'Important Events': 0,
    'Events/Second': lambda s: (
        (s['Important Events'] / (time.time() - s['Start Time']))),
    })
...
for event in events:
    ...
    # Collect stats
    if mystats.get('Enabled', False):
        mystats['Important Events'] += 1
```

To report statistics:

```
root.cpstats = cpstats.StatsPage()
```

To format statistics reports:

```
See 'Reporting', above.
```

class cherrypy.lib.cpstats.**ByteCountWrapper**(*rfile*)
Bases: object

Wraps a file-like object, counting the number of bytes read.

close()

next()

read(*size=- 1*)

readline(*size=- 1*)

readlines(*sizehint=0*)

class cherrypy.lib.cpstats.**StatsPage**
Bases: object

data()

formatting = {'CherryPy Applications': {'Bytes Read/Request': '%.3f', 'Bytes Read/Se

get_dict_collection(*v*, *formatting*)
Return ([headers], [rows]) for the given collection.

get_list_collection(*v*, *formatting*)
Return ([headers], [subrows]) for the given collection.

get_namespaces()
Yield (title, scalars, collections) for each namespace.

index()

pause(*namespace*)

resume(*namespace*)

class cherrypy.lib.cpstats.**StatsTool**
Bases: *cherrypy._cptools.Tool*

Record various information about the current request.

_setup()
Hook this tool into cherrypy.request.

The standard CherryPy request object will automatically call this method when the tool is "turned on" in config.

record_start()
Record the beginning of a request.

record_stop(*uriset=None*, *slow_queries=1.0*, *slow_queries_count=100*, *debug=False*, ***kwargs*)
Record the end of a request.

cherrypy.lib.cpstats.**_get_threading_ident**()

cherrypy.lib.cpstats.**average_uriset_time**(*s*)

cherrypy.lib.cpstats.**extrapolate_statistics**(*scope*)
Return an extrapolated copy of the given scope.

cherrypy.lib.cpstats.**iso_format**(*v*)

cherrypy.lib.cpstats.**locale_date**(*v*)

cherrypy.lib.cpstats.**pause_resume**(*ns*)

cherrypy.lib.cpstats.**proc_time**(*s*)

cherrypy.lib.cptools module

Functions for builtin CherryPy tools.

class cherrypy.lib.cptools.**MonitoredHeaderMap**
Bases: *cherrypy.lib.httputil.HeaderMap*

transform_key(*key*)

class cherrypy.lib.cptools.**SessionAuth**
Bases: object

Assert that the user is logged in.

_debug_message(*template*, *context={}*)

anonymous()
Provide a temporary user name for anonymous users.

check_username_and_password(*username*, *password*)

debug = False

do_check()
Assert username. Raise redirect, or return True if request handled.

do_login(*username*, *password*, *from_page='..'*, ***kwargs*)
Login. May raise redirect, or return True if request handled.

do_logout(*from_page='..'*, ***kwargs*)
Logout. May raise redirect, or return True if request handled.

login_screen(*from_page='..'*, *username=''*, *error_msg=''*, ***kwargs*)

on_check(*username*)

on_login(*username*)

on_logout(*username*)

run()

session_key = 'username'

cherrypy.lib.cptools.**accept**(*media=None*, *debug=False*)
Return the client's preferred media-type (from the given Content-Types).

If 'media' is None (the default), no test will be performed.

If 'media' is provided, it should be the Content-Type value (as a string) or values (as a list or tuple of strings) which the current resource can emit. The client's acceptable media ranges (as declared in the Accept request header) will be matched in order to these Content-Type values; the first such string is returned. That is, the return value will always be one of the strings provided in the 'media' arg (or None if 'media' is None).

If no match is found, then HTTPError 406 (Not Acceptable) is raised. Note that most web browsers send */* as a (low-quality) acceptable media range, which should match any Content-Type. In addition, "... if no Accept header field is present, then it is assumed that the client accepts all media types."

Matching types are checked in order of client preference first, and then in the order of the given 'media' values.

Note that this function does not honor accept-params (other than "q").

cherrypy.lib.cptools.**allow**(*methods=None*, *debug=False*)
Raise 405 if request.method not in methods (default ['GET', 'HEAD']).

The given methods are case-insensitive, and may be in any order. If only one method is allowed, you may supply a single string; if more than one, supply a list of strings.

Regardless of whether the current method is allowed or not, this also emits an 'Allow' response header, containing the given methods.

cherrypy.lib.cptools.**autovary**(*ignore=None*, *debug=False*)
Auto-populate the Vary response header based on request.header access.

cherrypy.lib.cptools.**convert_params**(*exception=<class 'ValueError'>*, *error=400*)
Convert request params based on function annotations, with error handling.

exception Exception class to catch.

status The HTTP error code to return to the client on failure.

cherrypy.lib.cptools.**flatten**(*debug=False*)
Wrap response.body in a generator that recursively iterates over body.

This allows cherrypy.response.body to consist of 'nested generators'; that is, a set of generators that yield generators.

cherrypy.lib.cptools.**ignore_headers**(*headers=('Range')*, *debug=False*)
Delete request headers whose field names are included in 'headers'.

This is a useful tool for working behind certain HTTP servers; for example, Apache duplicates the work that CP does for 'Range' headers, and will doubly-truncate the response.

cherrypy.lib.cptools.**log_hooks**(*debug=False*)
Write request.hooks to the cherrypy error log.

cherrypy.lib.cptools.**log_request_headers**(*debug=False*)
Write request headers to the cherrypy error log.

cherrypy.lib.cptools.**log_traceback**(*severity=40*, *debug=False*)
Write the last error's traceback to the cherrypy error log.

cherrypy.lib.cptools.**proxy**(*base=None*, *local='X-Forwarded-Host'*, *remote='X-Forwarded-For'*, *scheme='X-Forwarded-Proto'*, *debug=False*)
Change the base URL (scheme://host[:port][/path]).

For running a CP server behind Apache, lighttpd, or other HTTP server.

For Apache and lighttpd, you should leave the 'local' argument at the default value of 'X-Forwarded-Host'. For Squid, you probably want to set tools.proxy.local = 'Origin'.

If you want the new request.base to include path info (not just the host), you must explicitly set base to the full base path, and ALSO set 'local' to '', so that the X-Forwarded-Host request header (which never includes path info) does not override it. Regardless, the value for 'base' MUST NOT end in a slash.

cherrypy.request.remote.ip (the IP address of the client) will be rewritten if the header specified by the 'remote' arg is valid. By default, 'remote' is set to 'X-Forwarded-For'. If you do not want to rewrite remote.ip, set the 'remote' arg to an empty string.

cherrypy.lib.cptools.**redirect**(*url=''*, *internal=True*, *debug=False*)
Raise InternalRedirect or HTTPRedirect to the given url.

cherrypy.lib.cptools.**referer**(*pattern*, *accept=True*, *accept_missing=False*, *error=403*, *message='Forbidden Referer header.'*, *debug=False*)
Raise HTTPError if Referer header does/does not match the given pattern.

pattern A regular expression pattern to test against the Referer.

accept If True, the Referer must match the pattern; if False, the Referer must NOT match the pattern.

accept_missing If True, permit requests with no Referer header.

error The HTTP error code to return to the client on failure.

message A string to include in the response body on failure.

cherrypy.lib.cptools.**response_headers**(*headers=None*, *debug=False*)
Set headers on the response.

cherrypy.lib.cptools.**session_auth**(***kwargs*)

cherrypy.lib.cptools.**trailing_slash**(*missing=True*, *extra=False*, *status=None*, *debug=False*)
Redirect if path_info has (missing|extra) trailing slash.

cherrypy.lib.cptools.**validate_etags**(*autotags=False*, *debug=False*)
Validate the current ETag against If-Match, If-None-Match headers.

If autotags is True, an ETag response-header value will be provided from an MD5 hash of the response body (unless some other code has already provided an ETag header). If False (the default), the ETag will not be automatic.

WARNING: the autotags feature is not designed for URL's which allow methods other than GET. For example, if a POST to the same URL returns no content, the automatic ETag will be incorrect, breaking a fundamental use for entity tags in a possibly destructive fashion. Likewise, if you raise 304 Not Modified, the response body will be empty, the ETag hash will be incorrect, and your application will break. See **RFC 2616** Section 14.24.

cherrypy.lib.cptools.**validate_since**()
Validate the current Last-Modified against If-Modified-Since headers.

If no code has set the Last-Modified response header, then no validation will be performed.

cherrypy.lib.encoding module

class cherrypy.lib.encoding.**ResponseEncoder**(***kwargs*)
Bases: object

add_charset = True

debug = False

default_encoding = 'utf-8'

encode_stream(*encoding*)
Encode a streaming response body.

Use a generator wrapper, and just pray it works as the stream is being written out.

encode_string(*encoding*)
Encode a buffered response body.

encoding = None

errors = 'strict'

failmsg = 'Response body could not be encoded with %r.'

find_acceptable_charset()

text_only = True

class cherrypy.lib.encoding.**UTF8StreamEncoder**(*iterator*)
Bases: object

close()

next()

cherrypy.lib.encoding.**compress**(*body*, *compress_level*)
Compress 'body' at the given compress_level.

cherrypy.lib.encoding.**decode**(*encoding=None*, *default_encoding='utf-8'*)
Replace or extend the list of charsets used to decode a request entity.

Either argument may be a single string or a list of strings.

encoding If not None, restricts the set of charsets attempted while decoding a request entity to the given set (even if a different charset is given in the Content-Type request header).

default_encoding Only in effect if the 'encoding' argument is not given. If given, the set of charsets attempted while decoding a request entity is *extended* with the given value(s).

cherrypy.lib.encoding.**decompress**(*body*)

cherrypy.lib.encoding.**gzip**(*compress_level=5*, *mime_types=['text/html', 'text/plain']*, *debug=False*)
Try to gzip the response body if Content-Type in mime_types.

cherrypy.response.headers['Content-Type'] must be set to one of the values in the mime_types arg before calling this function.

The provided list of mime-types must be of one of the following form:

- `type/subtype`
- `type/*`
- `type/*+subtype`

No compression is performed if any of the following hold:

- The client sends no Accept-Encoding request header
- No 'gzip' or 'x-gzip' is present in the Accept-Encoding header
- No 'gzip' or 'x-gzip' with a qvalue > 0 is present
- The 'identity' value is given with a qvalue > 0.

cherrypy.lib.encoding.**prepare_iter**(*value*)
Ensure response body is iterable and resolves to False when empty.

cherrypy.lib.gctools module

class cherrypy.lib.gctools.**GCRoot**
Bases: `object`

A CherryPy page handler for testing reference leaks.

classes = [(<class 'cherrypy._cprequest.Request'>, 2, 2, 'Should be 1 in this request

index()

stats()

class cherrypy.lib.gctools.**ReferrerTree**(*ignore=None*, *maxdepth=2*, *maxparents=10*)
Bases: `object`

An object which gathers all referrers of an object to a given depth.

_format(*obj*, *descend=True*)
Return a string representation of a single object.

ascend(*obj*, *depth=1*)
Return a nested list containing referrers of the given object.

format(*tree*)
Return a list of string reprs from a nested list of referrers.

peek(*s*)
Return s, restricted to a sane length.

peek_length = 40

class cherrypy.lib.gctools.**RequestCounter**(*bus*)
Bases: *cherrypy.process.plugins.SimplePlugin*

after_request()

before_request()

start()

cherrypy.lib.gctools.**get_context**(*obj*)

cherrypy.lib.gctools.**get_instances**(*cls*)

cherrypy.lib.httputil module

HTTP library functions.

This module contains functions for building an HTTP application framework: any one, not just one whose name starts with "Ch". ;) If you reference any modules from some popular framework inside *this* module, FuManChu will personally hang you up by your thumbs and submit you to a public caning.

class cherrypy.lib.httputil.**AcceptElement**(*value*, *params=None*)
Bases: *cherrypy.lib.httputil.HeaderElement*

An element (with parameters) from an Accept* header's element list.

AcceptElement objects are comparable; the more-preferred object will be "less than" the less-preferred object. They are also therefore sortable; if you sort a list of AcceptElement objects, they will be listed in priority order; the most preferred value will be first. Yes, it should have been the other way around, but it's too late to fix now.

classmethod from_str(*elementstr*)
Construct an instance from a string of the form 'token;key=val'.

property qvalue
The qvalue, or priority, of this value.

class cherrypy.lib.httputil.**CaseInsensitiveDict**(**args*, ***kargs*)
Bases: jaraco.collections.KeyTransformingDict

A case-insensitive dict subclass.

Each key is changed on entry to title case.

static transform_key(*key*)

class cherrypy.lib.httputil.**HeaderElement**(*value*, *params=None*)
Bases: object

An element (with parameters) from an HTTP header's element list.

classmethod from_str(*elementstr*)
Construct an instance from a string of the form 'token;key=val'.

static parse(*elementstr*)
Transform 'token;key=val' to ('token', {'key': 'val'}).

class cherrypy.lib.httputil.**HeaderMap**(**args*, ***kargs*)
Bases: *cherrypy.lib.httputil.CaseInsensitiveDict*

A dict subclass for HTTP request and response headers.

Each key is changed on entry to str(key).title(). This allows headers to be case-insensitive and avoid duplicates.

Values are header values (decoded according to **RFC 2047** if necessary).

elements(*key*)
Return a sorted list of HeaderElements for the given header.

classmethod encode(*v*)
Return the given header name or value, encoded for HTTP output.

classmethod encode_header_item(*item*)

classmethod encode_header_items(*header_items*)
Prepare the sequence of name, value tuples into a form suitable for transmitting on the wire for HTTP.

encodings = ['ISO-8859-1']

output()
Transform self into a list of (name, value) tuples.

protocol = (1, 1)

use_rfc_2047 = True

values(*key*)
Return a sorted list of HeaderElement.value for the given header.

class cherrypy.lib.httputil.**Host**(*ip*, *port*, *name=None*)
Bases: object

An internet address.

name Should be the client's host name. If not available (because no DNS lookup is performed), the IP address should be used instead.

ip = '0.0.0.0'

name = 'unknown.tld'

port = 80

cherrypy.lib.httputil.**_parse_qs**(*qs*, *keep_blank_values=0*, *strict_parsing=0*, *encoding='utf-8'*)
Parse a query given as a string argument.

Arguments:

qs: URL-encoded query string to be parsed

keep_blank_values: flag indicating whether blank values in URL encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

strict_parsing: flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a ValueError exception.

Returns a dict, as G-d intended.

cherrypy.lib.httputil.**decode_TEXT**(*value*)
Decode RFC 2047 TEXT

```
>>> decode_TEXT("=?utf-8?q?f=C3=BCr?=") == b'f\xfcr'.decode('latin-1')
True
```

cherrypy.lib.httputil.**decode_TEXT_maybe**(*value*)
Decode the text but only if '=?' appears in it.

cherrypy.lib.httputil.**get_ranges**(*headervalue*, *content_length*)
Return a list of (start, stop) indices from a Range header, or None.

Each (start, stop) tuple will be composed of two ints, which are suitable for use in a slicing operation. That is, the header "Range: bytes=3-6", if applied against a Python string, is requesting resource[3:7]. This function will return the list [(3, 7)].

If this function returns an empty list, you should return HTTP 416.

cherrypy.lib.httputil.**header_elements**(*fieldname*, *fieldvalue*)
Return a sorted HeaderElement list from a comma-separated header string.

cherrypy.lib.httputil.**parse_query_string**(*query_string*, *keep_blank_values=True*, *encoding='utf-8'*)
Build a params dictionary from a query_string.

Duplicate key/value pairs in the provided query_string will be returned as {'key': [val1, val2, ...]}. Single key/values will be returned as strings: {'key': 'value'}.

cherrypy.lib.httputil.**protocol_from_http**(*protocol_str*)
Return a protocol tuple from the given 'HTTP/x.y' string.

cherrypy.lib.httputil.**urljoin**(**atoms*)
Return the given path *atoms, joined into a single URL.

This will correctly join a SCRIPT_NAME and PATH_INFO into the original URL, even if either atom is blank.

cherrypy.lib.httputil.**urljoin_bytes**(**atoms*)
Return the given path `*atoms`, joined into a single URL.

This will correctly join a SCRIPT_NAME and PATH_INFO into the original URL, even if either atom is blank.

cherrypy.lib.httputil.**valid_status**(*status*)
Return legal HTTP status Code, Reason-phrase and Message.

The status arg must be an int, a str that begins with an int or the constant from `http.client` stdlib module.

If status has no reason-phrase is supplied, a default reason- phrase will be provided.

```
>>> import http.client
>>> from http.server import BaseHTTPRequestHandler
>>> valid_status(http.client.ACCEPTED) == (
...     int(http.client.ACCEPTED),
... ) + BaseHTTPRequestHandler.responses[http.client.ACCEPTED]
True
```

cherrypy.lib.jsontools module

cherrypy.lib.jsontools.**json_handler**(**args*, ***kwargs*)

cherrypy.lib.jsontools.**json_in**(*content_type=['application/json', 'text/javascript'], force=True, debug=False, processor=<function json_processor>*)
Add a processor to parse JSON request entities: The default processor places the parsed data into request.json.

Incoming request entities which match the given content_type(s) will be deserialized from JSON to the Python equivalent, and the result stored at cherrypy.request.json. The 'content_type' argument may be a Content-Type string or a list of allowable Content-Type strings.

If the 'force' argument is True (the default), then entities of other content types will not be allowed; "415 Unsupported Media Type" is raised instead.

Supply your own processor to use a custom decoder, or to handle the parsed data differently. The processor can be configured via tools.json_in.processor or via the decorator method.

Note that the deserializer requires the client send a Content-Length request header, or it will raise "411 Length Required". If for any other reason the request entity cannot be deserialized from JSON, it will raise "400 Bad Request: Invalid JSON document".

cherrypy.lib.jsontools.**json_out**(*content_type='application/json', debug=False, handler=<function json_handler>*)
Wrap request.handler to serialize its output to JSON. Sets Content-Type.

If the given content_type is None, the Content-Type response header is not set.

Provide your own handler to use a custom encoder. For example cherrypy.config['tools.json_out.handler'] = <function>, or @json_out(handler=function).

cherrypy.lib.jsontools.**json_processor**(*entity*)
Read application/json data into request.json.

cherrypy.lib.locking module

class cherrypy.lib.locking.**LockChecker**(*session_id*, *timeout*)
Bases: object

Keep track of the time and detect if a timeout has expired

expired()

exception cherrypy.lib.locking.**LockTimeout**
Bases: Exception

An exception when a lock could not be acquired before a timeout period

class cherrypy.lib.locking.**NeverExpires**
Bases: object

expired()

class cherrypy.lib.locking.**Timer**(*expiration*)
Bases: object

A simple timer that will indicate when an expiration time has passed.

classmethod after(*elapsed*)
Return a timer that will expire after elapsed passes.

expired()

cherrypy.lib.profiler module

Profiler tools for CherryPy.

CherryPy users

You can profile any of your pages as follows:

```
from cherrypy.lib import profiler

class Root:
    p = profiler.Profiler("/path/to/profile/dir")

    @cherrypy.expose
    def index(self):
        self.p.run(self._index)

    def _index(self):
        return "Hello, world!"

cherrypy.tree.mount(Root())
```

You can also turn on profiling for all requests using the `make_app` function as WSGI middleware.

CherryPy developers

This module can be used whenever you make changes to CherryPy, to get a quick sanity-check on overall CP performance. Use the `--profile` flag when running the test suite. Then, use the `serve()` function to browse the results in a web browser. If you run this module from the command line, it will call `serve()` for you.

class cherrypy.lib.profiler.**ProfileAggregator**(*path=None*)
Bases: *cherrypy.lib.profiler.Profiler*

run(*func*, **args*, ***params*)
Dump profile data into self.path.

class cherrypy.lib.profiler.**Profiler**(*path=None*)
Bases: object

index()

menu()

report(*filename*)

run(*func*, **args*, ***params*)
Dump profile data into self.path.

statfiles()

Return type list of available profiles.

stats(*filename*, *sortby='cumulative'*)

Rtype stats(index) output of print_stats() for the given profile.

class cherrypy.lib.profiler.**make_app**(*nextapp*, *path=None*, *aggregate=False*)
Bases: object

cherrypy.lib.profiler.**new_func_strip_path**(*func_name*)
Make profiler output more readable by adding __init__ modules' parents

cherrypy.lib.profiler.**serve**(*path=None*, *port=8080*)

cherrypy.lib.reprconf module

Generic configuration system using unrepr.

Configuration data may be supplied as a Python dictionary, as a filename, or as an open file object. When you supply a filename or file, Python's builtin ConfigParser is used (with some extensions).

Namespaces

Configuration keys are separated into namespaces by the first "." in the key.

The only key that cannot exist in a namespace is the "environment" entry. This special entry 'imports' other config entries from a template stored in the Config.environments dict.

You can define your own namespaces to be called when new config is merged by adding a named handler to Config.namespaces. The name can be any string, and the handler must be either a callable or a context manager.

class cherrypy.lib.reprconf.**Config**(*file=None*, ***kwargs*)
Bases: dict

A dict-like set of configuration data, with defaults and namespaces.

May take a file, filename, or dict.

_apply(*config*)
Update self from a dict.

defaults = {}

environments = {}

namespaces = {'checker': <function <lambda>>, 'engine': <function _engine_namespace_

reset()
Reset self to default values.

update(*config*)
Update self from a dict, file, or filename.

class cherrypy.lib.reprconf.**NamespaceSet**
Bases: dict

A dict of config namespace names and handlers.

Each config entry should begin with a namespace name; the corresponding namespace handler will be called once for each config entry in that namespace, and will be passed two arguments: the config key (with the namespace removed) and the config value.

Namespace handlers may be any Python callable; they may also be context managers, in which case their __enter__ method should return a callable to be used as the handler. See cherrypy.tools (the Toolbox class) for an example.

copy() → a shallow copy of D

class cherrypy.lib.reprconf.**Parser**(*defaults=None*, *dict_type=<class 'collections.OrderedDict'>*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section='DEFAULT'*, *interpolation=<object object>*, *converters=<object object>*)
Bases: configparser.ConfigParser

Sub-class of ConfigParser that keeps the case of options and that raises an exception if the file cannot be read.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 48

_abc_registry = <_weakrefset.WeakSet object>

as_dict(*raw=False*, *vars=None*)
Convert an INI file to a dictionary

dict_from_file(*file*)

classmethod load(*input*)
Resolve 'input' to dict from a dict, file, or filename.

optionxform(*optionstr*)

read(*filenames*)
Read and parse a filename or an iterable of filenames.

Files that cannot be opened are silently ignored; this is designed so that you can specify an iterable of potential configuration file locations (e.g. current directory, user's home directory, systemwide directory), and all existing configuration files in the iterable will be read. A single filename may also be given.

Return list of successfully read files.

class cherrypy.lib.reprconf.**_Builder**
Bases: object

_build_call35(*o*)
Workaround for python 3.5 _ast.Call signature, docs found here https://greentreesnakes.readthedocs.org/en/latest/nodes.html

astnode(*s*)
Return a Python3 ast Node compiled from a string.

build(*o*)

build_Add(*o*)

build_Attribute(*o*)

build_BinOp(*o*)

build_Call(*o*)

build_Constant(*o*)

build_Dict(*o*)

build_Index(*o*)

build_List(*o*)

build_Mult(*o*)

build_Name(*o*)

build_NameConstant(*o*)

build_NoneType(*o*)

build_Num(*o*)

build_Str(*o*)

build_Subscript(*o*)

build_Tuple(*o*)

build_USub(*o*)

build_UnaryOp(*o*)

cherrypy.lib.reprconf.**attributes**(*full_attribute_name*)
Load a module and retrieve an attribute of that module.

cherrypy.lib.reprconf.**modules**(*modulePath*)
Load a module and retrieve a reference to that module.

cherrypy.lib.reprconf.**unrepr**(*s*)
Return a Python object compiled from a string.

cherrypy.lib.sessions module

Session implementation for CherryPy.

You need to edit your config file to use sessions. Here's an example:

```
[/]
tools.sessions.on = True
tools.sessions.storage_class = cherrypy.lib.sessions.FileSession
tools.sessions.storage_path = "/home/site/sessions"
tools.sessions.timeout = 60
```

This sets the session to be stored in files in the directory /home/site/sessions, and the session timeout to 60 minutes. If you omit `storage_class`, the sessions will be saved in RAM. `tools.sessions.on` is the only required line for working sessions, the rest are optional.

By default, the session ID is passed in a cookie, so the client's browser must have cookies enabled for your site.

To set data for the current session, use `cherrypy.session['fieldname'] = 'fieldvalue'`; to get data use `cherrypy.session.get('fieldname')`.

Locking sessions

By default, the `'locking'` mode of sessions is `'implicit'`, which means the session is locked early and unlocked late. Be mindful of this default mode for any requests that take a long time to process (streaming responses, expensive calculations, database lookups, API calls, etc), as other concurrent requests that also utilize sessions will hang until the session is unlocked.

If you want to control when the session data is locked and unlocked, set `tools.sessions.locking = 'explicit'`. Then call `cherrypy.session.acquire_lock()` and `cherrypy.session.release_lock()`. Regardless of which mode you use, the session is guaranteed to be unlocked when the request is complete.

Expiring Sessions

You can force a session to expire with *`cherrypy.lib.sessions.expire()`*. Simply call that function at the point you want the session to expire, and it will cause the session cookie to expire client-side.

Session Fixation Protection

If CherryPy receives, via a request cookie, a session id that it does not recognize, it will reject that id and create a new one to return in the response cookie. This helps prevent session fixation attacks. However, CherryPy "recognizes" a session id by looking up the saved session data for that id. Therefore, if you never save any session data, **you will get a new session id for every request**.

A side effect of CherryPy overwriting unrecognised session ids is that if you have multiple, separate CherryPy applications running on a single domain (e.g. on different ports), each app will overwrite the other's session id because by default they use the same cookie name (`"session_id"`) but do not recognise each others sessions. It is therefore a good idea to use a different name for each, for example:

```
[/]
...
tools.sessions.name = "my_app_session_id"
```

Sharing Sessions

If you run multiple instances of CherryPy (for example via mod_python behind Apache prefork), you most likely cannot use the RAM session backend, since each instance of CherryPy will have its own memory space. Use a different backend instead, and verify that all instances are pointing at the same file or db location. Alternately, you might try a load balancer which makes sessions "sticky". Google is your friend, there.

Expiration Dates

The response cookie will possess an expiration date to inform the client at which point to stop sending the cookie back in requests. If the server time and client time differ, expect sessions to be unreliable. **Make sure the system time of your server is accurate**.

CherryPy defaults to a 60-minute session timeout, which also applies to the cookie which is sent to the client. Unfortunately, some versions of Safari ("4 public beta" on Windows XP at least) appear to have a bug in their parsing of the GMT expiration date–they appear to interpret the date as one hour in the past. Sixty minutes minus one hour is pretty close to zero, so you may experience this bug as a new session id for every request, unless the requests are less than one second apart. To fix, try increasing the session.timeout.

On the other extreme, some users report Firefox sending cookies after their expiration date, although this was on a system with an inaccurate system time. Maybe FF doesn't trust system time.

class cherrypy.lib.sessions.**FileSession**(*id=None*, ***kwargs*)

Bases: *cherrypy.lib.sessions.Session*

Implementation of the File backend for sessions

storage_path The folder where session data will be saved. Each session will be saved as pickle.dump(data, expiration_time) in its own file; the filename will be self.SESSION_PREFIX + self.id.

lock_timeout A timedelta or numeric seconds indicating how long to block acquiring a lock. If None (default), acquiring a lock will block indefinitely.

LOCK_SUFFIX = '.lock'

SESSION_PREFIX = 'session-'

_delete()

_exists()

_get_file_path()

_load(*path=None*)

_save(*expiration_time*)

acquire_lock(*path=None*)
Acquire an exclusive lock on the currently-loaded session data.

clean_up()
Clean up expired sessions.

pickle_protocol = 4

release_lock(*path=None*)
Release the lock on the currently-loaded session data.

classmethod setup(***kwargs*)
Set up the storage system for file-based sessions.

This should only be called once per process; this will be done automatically when using sessions.init (as the built-in Tool does).

class cherrypy.lib.sessions.**MemcachedSession**(*id=None*, ***kwargs*)
Bases: *cherrypy.lib.sessions.Session*

_delete()

_exists()

_load()

_save(*expiration_time*)

acquire_lock()
Acquire an exclusive lock on the currently-loaded session data.

locks = {}

mc_lock = <unlocked _thread.RLock object owner=0 count=0>

release_lock()
Release the lock on the currently-loaded session data.

servers = ['localhost:11211']

classmethod setup(***kwargs*)
Set up the storage system for memcached-based sessions.

This should only be called once per process; this will be done automatically when using sessions.init (as the built-in Tool does).

class cherrypy.lib.sessions.**RamSession**(*id=None*, ***kwargs*)
Bases: *cherrypy.lib.sessions.Session*

_delete()

_exists()

_load()

_save(*expiration_time*)

acquire_lock()
Acquire an exclusive lock on the currently-loaded session data.

cache = {}

clean_up()
Clean up expired sessions.

locks = {}

release_lock()
Release the lock on the currently-loaded session data.

class cherrypy.lib.sessions.**Session**(*id=None*, ***kwargs*)
Bases: object

A CherryPy dict-like Session object (one per request).

_id = None

_regenerate()

clean_freq = 5
The poll rate for expired session cleanup in minutes.

clean_thread = None
Class-level Monitor which calls self.clean_up.

clean_up()
Clean up expired sessions.

clear() → None. Remove all items from D.

debug = False
If True, log debug information.

delete()
Delete stored session data.

generate_id()
Return a new session id.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

property id
Return the current session id.

id_observers = None
A list of callbacks to which to pass new id's.

items() → list of D's (key, value) pairs, as 2-tuples.

keys() → list of D's keys.

load()
Copy stored session data into this session instance.

loaded = False
If True, data has been retrieved from storage. This should happen automatically on the first attempt to access session data.

locked = False
If True, this session instance has exclusive read/write access to session data.

missing = False
True if the session requested by the client did not exist.

now()
Generate the session specific concept of 'now'.

Other session providers can override this to use alternative, possibly timezone aware, versions of 'now'.

originalid = None
The session id passed by the client. May be missing or unsafe.

pop(*key*, *default=False*)
Remove the specified key and return the corresponding value. If key is not found, default is returned if given, otherwise KeyError is raised.

regenerate()
Replace the current session (with a new id).

regenerated = False
True if the application called session.regenerate(). This is not set by internal calls to regenerate the session id.

save()
Save session data.

setdefault($k[, d]$) → D.get(k,d), also set D[k]=d if k not in D.

timeout = 60
Number of minutes after which to delete session data.

update(E) → None. Update D from E: for k in E: D[k] = E[k].

values() → list of D's values.

cherrypy.lib.sessions.**_add_MSIE_max_age_workaround**(*cookie*, *timeout*)
We'd like to use the "max-age" param as indicated in http://www.faqs.org/rfcs/rfc2109.html but IE doesn't save it to disk and the session is lost if people close the browser. So we have to use the old "expires" ... sigh ...

cherrypy.lib.sessions.**close**()
Close the session object for this request.

cherrypy.lib.sessions.**expire**()
Expire the current session cookie.

cherrypy.lib.sessions.**init**(*storage_type=None*, *path=None*, *path_header=None*, *name='session_id'*, *timeout=60*, *domain=None*, *secure=False*, *clean_freq=5*, *persistent=True*, *httponly=False*, *debug=False*, ***kwargs*)
Initialize session object (using cookies).

storage_class The Session subclass to use. Defaults to RamSession.

storage_type (deprecated) One of 'ram', 'file', memcached'. This will be used to look up the corresponding class in cherrypy.lib.sessions globals. For example, 'file' will use the FileSession class.

path The 'path' value to stick in the response cookie metadata.

path_header If 'path' is None (the default), then the response cookie 'path' will be pulled from request.headers[path_header].

name The name of the cookie.

timeout The expiration timeout (in minutes) for the stored session data. If 'persistent' is True (the default), this is also the timeout for the cookie.

domain The cookie domain.

secure If False (the default) the cookie 'secure' value will not be set. If True, the cookie 'secure' value will be set (to 1).

clean_freq (minutes) The poll rate for expired session cleanup.

persistent If True (the default), the 'timeout' argument will be used to expire the cookie. If False, the cookie will not have an expiry, and the cookie will be a "session cookie" which expires when the browser is closed.

httponly If False (the default) the cookie 'httponly' value will not be set. If True, the cookie 'httponly' value will be set (to 1).

Any additional kwargs will be bound to the new Session instance, and may be specific to the storage type. See the subclass of Session you're using for more information.

`cherrypy.lib.sessions.`**`save`**`()`
Save any changed session data.

`cherrypy.lib.sessions.`**`set_response_cookie`**(*path=None*, *path_header=None*, *name='session_id'*, *timeout=60*, *domain=None*, *secure=False*, *httponly=False*)
Set a response cookie for the client.

path the 'path' value to stick in the response cookie metadata.

path_header if 'path' is None (the default), then the response cookie 'path' will be pulled from request.headers[path_header].

name the name of the cookie.

timeout the expiration timeout for the cookie. If 0 or other boolean False, no 'expires' param will be set, and the cookie will be a "session cookie" which expires when the browser is closed.

domain the cookie domain.

secure if False (the default) the cookie 'secure' value will not be set. If True, the cookie 'secure' value will be set (to 1).

httponly If False (the default) the cookie 'httponly' value will not be set. If True, the cookie 'httponly' value will be set (to 1).

cherrypy.lib.static module

Module with helpers for serving static files.

`cherrypy.lib.static.`**`_attempt`**(*filename*, *content_types*, *debug=False*)

`cherrypy.lib.static.`**`_make_content_disposition`**(*disposition*, *file_name*)
Create HTTP header for downloading a file with a UTF-8 filename.

This function implements the recommendations of **RFC 6266#appendix-D**. See this and related answers: https://stackoverflow.com/a/8996249/2173868.

`cherrypy.lib.static.`**`_serve_fileobj`**(*fileobj*, *content_type*, *content_length*, *debug=False*)
Internal. Set response.body to the given file object, perhaps ranged.

`cherrypy.lib.static.`**`_setup_mimetypes`**`()`
Pre-initialize global mimetype map.

`cherrypy.lib.static.`**`serve_download`**(*path*, *name=None*)
Serve 'path' as an application/x-download attachment.

`cherrypy.lib.static.`**`serve_file`**(*path*, *content_type=None*, *disposition=None*, *name=None*, *debug=False*)
Set status, headers, and body in order to serve the given path.

The Content-Type header will be set to the content_type arg, if provided. If not provided, the Content-Type will be guessed by the file extension of the 'path' argument.

If disposition is not None, the Content-Disposition header will be set to "<disposition>; filename=<name>; filename*=utf-8"<name>" as described in RFC 6266#appendix-D. If name is None, it will be set to the basename of path. If disposition is None, no Content-Disposition header will be written.

cherrypy.lib.static.**serve_fileobj**(*fileobj*, *content_type=None*, *disposition=None*, *name=None*, *debug=False*)

Set status, headers, and body in order to serve the given file object.

The Content-Type header will be set to the content_type arg, if provided.

If disposition is not None, the Content-Disposition header will be set to "<disposition>; filename=<name>; filename*=utf-8"<name>" as described in RFC 6266#appendix-D. If name is None, 'filename' will not be set. If disposition is None, no Content-Disposition header will be written.

CAUTION: If the request contains a 'Range' header, one or more seek()s will be performed on the file object. This may cause undesired behavior if the file object is not seekable. It could also produce undesired results if the caller set the read position of the file object prior to calling serve_fileobj(), expecting that the data would be served starting from that position.

cherrypy.lib.static.**staticdir**(*section*, *dir*, *root=''*, *match=''*, *content_types=None*, *index=''*, *debug=False*)

Serve a static resource from the given (root +) dir.

match If given, request.path_info will be searched for the given regular expression before attempting to serve static content.

content_types If given, it should be a Python dictionary of {file-extension: content-type} pairs, where 'file-extension' is a string (e.g. "gif") and 'content-type' is the value to write out in the Content-Type response header (e.g. "image/gif").

index If provided, it should be the (relative) name of a file to serve for directory requests. For example, if the dir argument is '/home/me', the Request-URI is 'myapp', and the index arg is 'index.html', the file '/home/me/myapp/index.html' will be sought.

cherrypy.lib.static.**staticfile**(*filename*, *root=None*, *match=''*, *content_types=None*, *debug=False*)

Serve a static resource from the given (root +) filename.

match If given, request.path_info will be searched for the given regular expression before attempting to serve static content.

content_types If given, it should be a Python dictionary of {file-extension: content-type} pairs, where 'file-extension' is a string (e.g. "gif") and 'content-type' is the value to write out in the Content-Type response header (e.g. "image/gif").

cherrypy.lib.xmlrpcutil module

XML-RPC tool helpers.

cherrypy.lib.xmlrpcutil.**_set_response**(*body*)

Set up HTTP status, headers and body within CherryPy.

cherrypy.lib.xmlrpcutil.**on_error**(**args*, ***kwargs*)

Construct HTTP response body for an error response.

cherrypy.lib.xmlrpcutil.**patched_path**(*path*)

Return 'path', doctored for RPC.

cherrypy.lib.xmlrpcutil.**process_body**()

Return (params, method) from request body.

cherrypy.lib.xmlrpcutil.**respond**(*body*, *encoding='utf-8'*, *allow_none=0*)
Construct HTTP response body.

Module contents

CherryPy Library.

class cherrypy.lib.**file_generator**(*input*, *chunkSize=65536*)
Bases: object

Yield the given input (a file object) in chunks (default 64k).

(Core)

next()
Return next chunk of file.

cherrypy.lib.**file_generator_limited**(*fileobj*, *count*, *chunk_size=65536*)
Yield the given file object in chunks.

Stopps after *count* bytes has been emitted. Default chunk size is 64kB. (Core)

cherrypy.lib.**is_closable_iterator**(*obj*)
Detect if the given object is both closable and iterator.

cherrypy.lib.**is_iterator**(*obj*)
Detect if the object provided implements the iterator protocol.

(i.e. like a generator).

This will return False for objects which are iterable, but not iterators themselves.

cherrypy.lib.**set_vary_header**(*response*, *header_name*)
Add a Vary header to a response.

cherrypy.process package

Submodules

cherrypy.process.plugins module

Site services for use with a Web Site Process Bus.

class cherrypy.process.plugins.**Autoreloader**(*bus*, *frequency=1*, *match='.*'*)
Bases: *cherrypy.process.plugins.Monitor*

Monitor which re-executes the process when files change.

This plugin restarts the process (via os.execv()) if any of the files it monitors change (or is deleted). By default, the autoreloader monitors all imported modules; you can add to the set by adding to autoreload.files:

```
cherrypy.engine.autoreload.files.add(myFile)
```

If there are imported files you do *not* wish to monitor, you can adjust the match attribute, a regular expression. For example, to stop monitoring cherrypy itself:

```
cherrypy.engine.autoreload.match = r'^(?!cherrypy).+'
```

Like all *Monitor* plugins, the autoreload plugin takes a `frequency` argument. The default is 1 second; that is, the autoreloader will examine files once each second.

static _archive_for_zip_module(*module*)
Return the archive filename for the module if relevant.

classmethod _file_for_file_module(*module*)
Return the file for the module.

classmethod _file_for_module(*module*)
Return the relevant file for the module.

static _make_absolute(*filename*)
Ensure filename is absolute to avoid effect of os.chdir.

files = None
The set of files to poll for modifications.

frequency = 1
The interval in seconds at which to poll for modified files.

match = '.*'
A regular expression by which to match filenames.

run()
Reload the process if registered files have been modified.

start()
Start our own background task thread for self.run.

sysfiles()
Return a Set of sys.modules filenames to monitor.

class cherrypy.process.plugins.**BackgroundTask**(*interval*, *function*, *args=[]*, *kwargs={}*, *bus=None*)

Bases: threading.Thread

A subclass of threading.Thread whose run() method repeats.

Use this class for most repeating tasks. It uses time.sleep() to wait for each interval, which isn't very responsive; that is, even if you call self.cancel(), you'll have to wait until the sleep() call finishes before the thread stops. To compensate, it defaults to being daemonic, which means it won't delay stopping the whole process.

cancel()

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class cherrypy.process.plugins.**Daemonizer**(*bus*, *stdin='/dev/null'*, *stdout='/dev/null'*, *stderr='/dev/null'*)

Bases: *cherrypy.process.plugins.SimplePlugin*

Daemonize the running script.

Use this with a Web Site Process Bus via:

```
Daemonizer(bus).subscribe()
```

When this component finishes, the process is completely decoupled from the parent environment. Please note that when this component is used, the return code from the parent process will still be 0 if a startup error occurs

in the forked children. Errors in the initial daemonizing process still return proper exit codes. Therefore, if you use this plugin to daemonize, don't use the return code as an accurate indicator of whether the process fully started. In fact, that return code only indicates if the process successfully finished the first fork.

static daemonize(*stdin='/dev/null'*, *stdout='/dev/null'*, *stderr='/dev/null'*, *logger=<function Daemonizer.<lambda>>*)

start()

class cherrypy.process.plugins.**DropPrivileges**(*bus*, *umask=None*, *uid=None*, *gid=None*)
Bases: *cherrypy.process.plugins.SimplePlugin*

Drop privileges. uid/gid arguments not available on Windows.

Special thanks to Gavin Baker

property gid
Unix.

Type The gid under which to run. Availability

start()

property uid
Unix.

Type The uid under which to run. Availability

property umask
The default permission mode for newly created files and directories.

Usually expressed in octal format, for example, `0644`. Availability: Unix, Windows.

class cherrypy.process.plugins.**Monitor**(*bus*, *callback*, *frequency=60*, *name=None*)
Bases: *cherrypy.process.plugins.SimplePlugin*

WSPBus listener to periodically run a callback in its own thread.

callback = None
The function to call at intervals.

frequency = 60
The time in seconds between callback runs.

graceful()
Stop the callback's background task thread and restart it.

start()
Start our callback in its own background thread.

stop()
Stop our callback's background task thread.

thread = None
A *BackgroundTask* thread.

class cherrypy.process.plugins.**PIDFile**(*bus*, *pidfile*)
Bases: *cherrypy.process.plugins.SimplePlugin*

Maintain a PID file via a WSPBus.

exit()

start()

class cherrypy.process.plugins.**PerpetualTimer**(**args*, ***kwargs*)
Bases: threading.Timer

A responsive subclass of threading.Timer whose run() method repeats.

Use this timer only when you really need a very interruptible timer; this checks its 'finished' condition up to 20 times a second, which can results in pretty high CPU usage

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class cherrypy.process.plugins.**SignalHandler**(*bus*)
Bases: object

Register bus channels (and listeners) for system signals.

You can modify what signals your application listens for, and what it does when it receives signals, by modifying *SignalHandler.handlers*, a dict of {signal name: callback} pairs. The default set is:

```
handlers = {'SIGTERM': self.bus.exit,
            'SIGHUP': self.handle_SIGHUP,
            'SIGUSR1': self.bus.graceful,
           }
```

The SignalHandler.handle_SIGHUP`() method calls *bus.restart()* if the process is daemonized, but *bus.exit()* if the process is attached to a TTY. This is because Unix window managers tend to send SIGHUP to terminal windows when the user closes them.

Feel free to add signals which are not available on every platform. The *SignalHandler* will ignore errors raised from attempting to register handlers for unknown signals.

_handle_signal(*signum=None*, *frame=None*)
Python signal handler (self.set_handler subscribes it for you).

_is_daemonized()
Return boolean indicating if the current process is running as a daemon.

The criteria to determine the *daemon* condition is to verify if the current pid is not the same as the one that got used on the initial construction of the plugin *and* the stdin is not connected to a terminal.

The sole validation of the tty is not enough when the plugin is executing inside other process like in a CI tool (Buildbot, Jenkins).

_jython_SIGINT_handler(*signum=None*, *frame=None*)

handle_SIGHUP()
Restart if daemonized, else exit.

handlers = {}
A map from signal names (e.g. 'SIGTERM') to handlers (e.g. bus.exit).

set_handler(*signal*, *listener=None*)
Subscribe a handler for the given signal (number or name).

If the optional 'listener' argument is provided, it will be subscribed as a listener for the given signal's channel.

If the given signal name or number is not available on the current platform, ValueError is raised.

signals = {<Signals.SIGHUP: 1>: 'SIGHUP', <Signals.SIGINT: 2>: 'SIGINT', <Signals.SI
A map from signal numbers to names.

subscribe()
Subscribe self.handlers to signals.

unsubscribe()
Unsubscribe self.handlers from signals.

class cherrypy.process.plugins.**SimplePlugin**(*bus*)
Bases: object

Plugin base class which auto-subscribes methods for known channels.

bus = None
A *Bus*, usually cherrypy.engine.

subscribe()
Register this object as a (multi-channel) listener on the bus.

unsubscribe()
Unregister this object as a listener on the bus.

class cherrypy.process.plugins.**ThreadManager**(*bus*)
Bases: *cherrypy.process.plugins.SimplePlugin*

Manager for HTTP request threads.

If you have control over thread creation and destruction, publish to the 'acquire_thread' and 'release_thread' channels (for each thread). This will register/unregister the current thread and publish to 'start_thread' and 'stop_thread' listeners in the bus as needed.

If threads are created and destroyed by code you do not control (e.g., Apache), then, at the beginning of every HTTP request, publish to 'acquire_thread' only. You should not publish to 'release_thread' in this case, since you do not know whether the thread will be re-used or not. The bus will call 'stop_thread' listeners for you when it stops.

acquire_thread()
Run 'start_thread' listeners for the current thread.

If the current thread has already been seen, any 'start_thread' listeners will not be run again.

graceful()
Release all threads and run all 'stop_thread' listeners.

release_thread()
Release the current thread and run 'stop_thread' listeners.

stop()
Release all threads and run all 'stop_thread' listeners.

threads = None
index number} pairs.

Type A map of {thread ident

cherrypy.process.servers module

Starting in CherryPy 3.1, cherrypy.server is implemented as an Engine Plugin. It's an instance of *cherrypy._cpserver.Server*, which is a subclass of *cherrypy.process.servers.ServerAdapter*. The ServerAdapter class is designed to control other servers, as well.

Multiple servers/ports

If you need to start more than one HTTP server (to serve on multiple ports, or protocols, etc.), you can manually register each one and then start them all with engine.start:

```
s1 = ServerAdapter(
    cherrypy.engine,
    MyWSGIServer(host='0.0.0.0', port=80)
)
s2 = ServerAdapter(
    cherrypy.engine,
    another.HTTPServer(host='127.0.0.1', SSL=True)
)
s1.subscribe()
s2.subscribe()
cherrypy.engine.start()
```

FastCGI/SCGI

There are also FlupFCGIServer and FlupSCGIServer classes in *cherrypy.process.servers*. To start an fcgi server, for example, wrap an instance of it in a ServerAdapter:

```
addr = ('0.0.0.0', 4000)
f = servers.FlupFCGIServer(application=cherrypy.tree, bindAddress=addr)
s = servers.ServerAdapter(cherrypy.engine, httpserver=f, bind_addr=addr)
s.subscribe()
```

The cherryd startup script will do the above for you via its `-f` flag. Note that you need to download and install flup yourself, whether you use `cherryd` or not.

FastCGI

A very simple setup lets your cherry run with FastCGI. You just need the flup library, plus a running Apache server (with `mod_fastcgi`) or lighttpd server.

CherryPy code

hello.py:

```
#!/usr/bin/python
import cherrypy

class HelloWorld:
    '''Sample request handler class.'''
    @cherrypy.expose
```

(continues on next page)

(continued from previous page)

```
    def index(self):
        return "Hello world!"

cherrypy.tree.mount(HelloWorld())
# CherryPy autoreload must be disabled for the flup server to work
cherrypy.config.update({'engine.autoreload.on':False})
```

Then run /deployguide/cherryd with the '-f' arg:

```
cherryd -c <myconfig> -d -f -i hello.py
```

Apache

At the top level in httpd.conf:

```
FastCgiIpcDir /tmp
FastCgiServer /path/to/cherry.fcgi -idle-timeout 120 -processes 4
```

And inside the relevant VirtualHost section:

```
# FastCGI config
AddHandler fastcgi-script .fcgi
ScriptAliasMatch (.*$) /path/to/cherry.fcgi$1
```

Lighttpd

For Lighttpd you can follow these instructions. Within `lighttpd.conf` make sure `mod_fastcgi` is active within `server.modules`. Then, within your `$HTTP["host"]` directive, configure your fastcgi script like the following:

```
$HTTP["url"] =~ "" {
  fastcgi.server = (
    "/" => (
      "script.fcgi" => (
        "bin-path" => "/path/to/your/script.fcgi",
        "socket"          => "/tmp/script.sock",
        "check-local"     => "disable",
        "disable-time"    => 1,
        "min-procs"       => 1,
        "max-procs"       => 1, # adjust as needed
      ),
    ),
  )
} # end of $HTTP["url"] =~ "^/"
```

Please see Lighttpd FastCGI Docs for an explanation of the possible configuration options.

class cherrypy.process.servers.**FlupCGIServer**(**args*, ***kwargs*)

Bases: object

Adapter for a flup.server.cgi.WSGIServer.

start()

Start the CGI server.

stop()
Stop the HTTP server.

class cherrypy.process.servers.**FlupFCGIServer**(**args*, ***kwargs*)
Bases: object

Adapter for a flup.server.fcgi.WSGIServer.

start()
Start the FCGI server.

stop()
Stop the HTTP server.

class cherrypy.process.servers.**FlupSCGIServer**(**args*, ***kwargs*)
Bases: object

Adapter for a flup.server.scgi.WSGIServer.

start()
Start the SCGI server.

stop()
Stop the HTTP server.

class cherrypy.process.servers.**ServerAdapter**(*bus*, *httpserver=None*, *bind_addr=None*)
Bases: object

Adapter for an HTTP server.

If you need to start more than one HTTP server (to serve on multiple ports, or protocols, etc.), you can manually register each one and then start them all with bus.start:

```
s1 = ServerAdapter(bus, MyWSGIServer(host='0.0.0.0', port=80))
s2 = ServerAdapter(bus, another.HTTPServer(host='127.0.0.1', SSL=True))
s1.subscribe()
s2.subscribe()
bus.start()
```

_get_base()

_start_http_thread()
HTTP servers MUST be running in new threads, so that the main thread persists to receive KeyboardInterrupt's. If an exception is raised in the httpserver's thread then it's trapped here, and the bus (and therefore our httpserver) are shut down.

property bound_addr
The bind address, or if it's an ephemeral port and the socket has been bound, return the actual port bound.

property description
A description about where this server is bound.

restart()
Restart the HTTP server.

start()
Start the HTTP server.

stop()
Stop the HTTP server.

subscribe()

unsubscribe()

wait()
Wait until the HTTP server is ready to receive requests.

class cherrypy.process.servers.**Timeouts**
Bases: object

free = 1

occupied = 5

cherrypy.process.servers.**_safe_wait**(*host*, *port*)
On systems where a loopback interface is not available and the server is bound to all interfaces, it's difficult to determine whether the server is in fact occupying the port. In this case, just issue a warning and move on. See issue #1100.

cherrypy.process.win32 module

Windows service. Requires pywin32.

class cherrypy.process.win32.**ConsoleCtrlHandler**(*bus*)
Bases: *cherrypy.process.plugins.SimplePlugin*

A WSPBus plugin for handling Win32 console events (like Ctrl-C).

handle(*event*)
Handle console control events (like Ctrl-C).

start()

stop()

class cherrypy.process.win32.**Win32Bus**
Bases: *cherrypy.process.wspbus.Bus*

A Web Site Process Bus implementation for Win32.

Instead of time.sleep, this bus blocks using native win32event objects.

_get_state_event(*state*)
Return a win32event for the given state (creating it if needed).

property state

wait(*state*, *interval=0.1*, *channel=None*)
Wait for the given state(s), KeyboardInterrupt or SystemExit.

Since this class uses native win32event objects, the interval argument is ignored.

class cherrypy.process.win32.**_ControlCodes**
Bases: dict

Control codes used to "signal" a service via ControlService.

User-defined control codes are in the range 128-255. We generally use the standard Python value for the Linux signal and add 128. Example:

```
>>> signal.SIGUSR1
10
control_codes['graceful'] = 128 + 10
```

key_for(*obj*)
For the given value, return its corresponding key.

cherrypy.process.win32.**signal_child**(*service*, *command*)

cherrypy.process.wspbus module

An implementation of the Web Site Process Bus.

This module is completely standalone, depending only on the stdlib.

Web Site Process Bus

A Bus object is used to contain and manage site-wide behavior: daemonization, HTTP server start/stop, process reload, signal handling, drop privileges, PID file management, logging for all of these, and many more.

In addition, a Bus object provides a place for each web framework to register code that runs in response to site-wide events (like process start and stop), or which controls or otherwise interacts with the site-wide components mentioned above. For example, a framework which uses file-based templates would add known template filenames to an autoreload component.

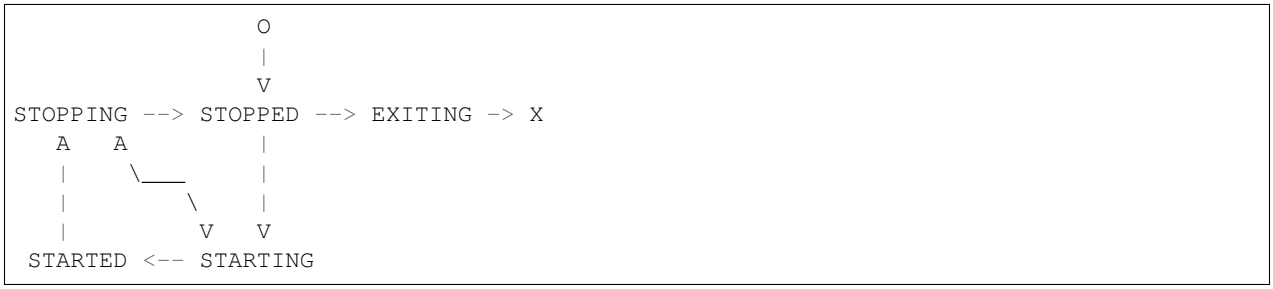
Ideally, a Bus object will be flexible enough to be useful in a variety of invocation scenarios:

1. The deployer starts a site from the command line via a framework-neutral deployment script; applications from multiple frameworks are mixed in a single site. Command-line arguments and configuration files are used to define site-wide components such as the HTTP server, WSGI component graph, autoreload behavior, signal handling, etc.
2. The deployer starts a site via some other process, such as Apache; applications from multiple frameworks are mixed in a single site. Autoreload and signal handling (from Python at least) are disabled.
3. The deployer starts a site via a framework-specific mechanism; for example, when running tests, exploring tutorials, or deploying single applications from a single framework. The framework controls which site-wide components are enabled as it sees fit.

The Bus object in this package uses topic-based publish-subscribe messaging to accomplish all this. A few topic channels are built in ('start', 'stop', 'exit', 'graceful', 'log', and 'main'). Frameworks and site containers are free to define their own. If a message is sent to a channel that has not been defined or has no listeners, there is no effect.

In general, there should only ever be a single Bus object per process. Frameworks and site containers share a single Bus object by publishing messages and subscribing listeners.

The Bus object works as a finite state machine which models the current state of the process. Bus methods move it from one state to another; those methods then publish to subscribed listeners on the channel for the new state.:



class cherrypy.process.wspbus.**Bus**

Bases: object

Process state-machine and messenger for HTTP site deployment.

All listeners for a given channel are guaranteed to be called even if others at the same channel fail. Each failure is logged, but execution proceeds on to the next listener. The only way to stop all processing from inside a listener is to raise SystemExit and stop the whole server.

_clean_exit()
Assert that the Bus is not running in atexit handler callback.

_do_execv()
Re-execute the current process.

This must be called from the main thread, because certain platforms (OS X) don't allow execv to be called in a child thread very well.

static _extend_pythonpath(*env*)
Prepend current working dir to PATH environment variable if needed.

If sys.path[0] is an empty string, the interpreter was likely invoked with -m and the effective path is about to change on re-exec. Add the current directory to $PYTHONPATH to ensure that the new process sees the same path.

This issue cannot be addressed in the general case because Python cannot reliably reconstruct the original command line (http://bugs.python.org/issue14208).

(This idea filched from tornado.autoreload)

static _get_interpreter_argv()
Retrieve current Python interpreter's arguments.

Returns empty tuple in case of frozen mode, uses built-in arguments reproduction function otherwise.

Frozen mode is possible for the app has been packaged into a binary executable using py2exe. In this case the interpreter's arguments are already built-in into that executable.

Seealso https://github.com/cherrypy/cherrypy/issues/1526

Ref: https://pythonhosted.org/PyInstaller/runtime-information.html

static _get_true_argv()
Retrieve all real arguments of the python interpreter.

...even those not listed in `sys.argv`

Seealso http://stackoverflow.com/a/28338254/595220

Seealso http://stackoverflow.com/a/6683222/595220

Seealso http://stackoverflow.com/a/28414807/595220

_set_cloexec()
Set the CLOEXEC flag on all open files (except stdin/out/err).

If self.max_cloexec_files is an integer (the default), then on platforms which support it, it represents the max open files setting for the operating system. This function will be called just before the process is restarted via os.execv() to prevent open files from persisting into the new process.

Set self.max_cloexec_files to 0 to disable this behavior.

block(*interval=0.1*)
Wait for the EXITING state, KeyboardInterrupt or SystemExit.

This function is intended to be called only by the main thread. After waiting for the EXITING state, it also waits for all threads to terminate, and then calls os.execv if self.execv is True. This design allows another thread to call bus.restart, yet have the main thread perform the actual execv call (required on some platforms).

execv = False

exit()
Stop all services and prepare to exit the process.

graceful()
Advise all services to reload.

log(*msg=''*, *level=20*, *traceback=False*)
Log the given message. Append the last traceback if requested.

max_cloexec_files = 1048576

publish(*channel*, **args*, ***kwargs*)
Return output of all subscribers for the given channel.

restart()
Restart the process (may close connections).

This method does not restart the process from the calling thread; instead, it stops the bus and asks the main thread to call execv.

start()
Start all services.

start_with_callback(*func*, *args=None*, *kwargs=None*)
Start 'func' in a new thread T, then start self (and return T).

state = states.STOPPED

states = <cherrypy.process.wspbus._StateEnum object>

stop()
Stop all services.

subscribe(*channel*, *callback=None*, *priority=None*)
Add the given callback at the given channel (if not present).

If callback is None, return a partial suitable for decorating the callback.

unsubscribe(*channel*, *callback*)
Discard the given callback (if present).

wait(*state*, *interval=0.1*, *channel=None*)
Poll for the given state(s) at intervals; publish to channel.

exception cherrypy.process.wspbus.**ChannelFailures**(**args*, ***kwargs*)
Bases: Exception

Exception raised during errors on Bus.publish().

delimiter = '\n'

get_instances()
Return a list of seen exception instances.

handle_exception()
Append the current exception to self.

class cherrypy.process.wspbus.**_StateEnum**
Bases: object

class State
Bases: object

name = None

Module contents

Site container for an HTTP server.

A Web Site Process Bus object is used to connect applications, servers, and frameworks with site-wide services such as daemonization, process reload, signal handling, drop privileges, PID file management, logging for all of these, and many more.

The 'plugins' module defines a few abstract and concrete services for use with the bus. Some use tool-specific channels; see the documentation for each class.

cherrypy.scaffold package

Module contents

<MyProject>, a CherryPy application.

Use this as a base for creating new CherryPy applications. When you want to make a new app, copy and paste this folder to some other location (maybe site-packages) and rename it to the name of your project, then tweak as desired.

Even before any tweaking, this should serve a few demonstration pages. Change to this directory and run:

    cherryd -c site.conf

class cherrypy.scaffold.**Root**
Bases: object

Declaration of the CherryPy app URI structure.

_cp_config = {'tools.log_tracebacks.on': True}

default(**args*, ***kwargs*)
Render catch-all args and kwargs.

files(***kw*)

index()
Render HTML-template at the root path of the web-app.

other(*a=2*, *b='bananas'*, *c=None*)
Render number of fruits based on third argument.

cherrypy.test package

Submodules

cherrypy.test._test_decorators module

Test module for the @-decorator syntax, which is version-specific

class cherrypy.test._test_decorators.**ExposeExamples**
Bases: object

alias1()

alias2()

alias3()

andrews()

call_alias()

call_empty()

nesbitt()

no_call()

watson()

class cherrypy.test._test_decorators.**ToolExamples**
Bases: object

blah()

cherrypy.test._test_states_demo module

class cherrypy.test._test_states_demo.**Root**
Bases: object

exit()

index()

mtimes()

pid()

start()

cherrypy.test.benchmark module

CherryPy Benchmark Tool

Usage: benchmark.py [options]

–null: use a null Request object (to bench the HTTP server only) –notests: start the server but do not run the tests; this allows

you to check the tested pages with a browser

–help: show this help message –cpmodpy: run tests via apache on 54583 (with the builtin _cpmodpy) –modpython: run tests via apache on 54583 (with modpython_gateway) –ab=path: Use the ab script/executable at 'path' (see below) –apache=path: Use the apache script/exe at 'path' (see below)

To run the benchmarks, the Apache Benchmark tool "ab" must either be on your system path, or specified via the –ab=path option.

To run the modpython tests, the "apache" executable or script must be on your system path, or provided via the –apache=path option. On some platforms, "apache" may be called "apachectl" or "apache2ctl"–create a symlink to them if needed.

class cherrypy.test.benchmark.**ABSession**(*path='/cpbench/users/rdelon/apps/blog/hello'*, *requests=1000*, *concurrency=10*)
Bases: object

A session of 'ab', the Apache HTTP server benchmarking tool.

Example output from ab:

This is ApacheBench, Version 2.0.40-dev <$Revision: 1.121.2.1 $> apache-2.0 Copyright (c) 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/ Copyright (c) 1998-2002 The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient) Completed 100 requests Completed 200 requests Completed 300 requests Completed 400 requests Completed 500 requests Completed 600 requests Completed 700 requests Completed 800 requests Completed 900 requests

Server Software: CherryPy/3.1beta Server Hostname: 127.0.0.1 Server Port: 54583

Document Path: /static/index.html Document Length: 14 bytes

Concurrency Level: 10 Time taken for tests: 9.643867 seconds Complete requests: 1000 Failed requests: 0 Write errors: 0 Total transferred: 189000 bytes HTML transferred: 14000 bytes Requests per second: 103.69 [#/sec] (mean) Time per request: 96.439 [ms] (mean) Time per request: 9.644 [ms] (mean, across all concurrent requests) Transfer rate: 19.08 [Kbytes/sec] received

Connection Times (ms) min mean[+/-sd] median max

Connect: 0 0 2.9 0 10 Processing: 20 94 7.3 90 130 Waiting: 0 43 28.1 40 100 Total: 20 95 7.3 100 130

Percentage of the requests served within a certain time (ms)

50% 100 66% 100 75% 100 80% 100 90% 100 95% 100 98% 100 99% 110

100% 130 (longest request)

Finished 1000 requests

args()

parse_patterns = [('complete_requests', 'Completed', b'^Complete requests:\\s*(\\d+)')

run()

class cherrypy.test.benchmark.**Root**

Bases: object

hello()

index()

sizer(*size*)

cherrypy.test.benchmark.**print_report**(*rows*)

cherrypy.test.benchmark.**run_standard_benchmarks**()

cherrypy.test.benchmark.**size_report**(*sizes=(10, 100, 1000, 10000, 100000, 100000000)*, *concurrency=50*)

cherrypy.test.benchmark.**thread_report**(*path='/cpbench/users/rdelon/apps/blog/hello'*, *concurrency=(25, 50, 100, 200, 400)*)

cherrypy.test.checkerdemo module

Demonstration app for cherrypy.checker.

This application is intentionally broken and badly designed. To demonstrate the output of the CherryPy Checker, simply execute this module.

class cherrypy.test.checkerdemo.**Root**

Bases: object

cherrypy.test.helper module

A library of helper functions for the CherryPy test suite.

class cherrypy.test.helper.**CPProcess**(*wait=False*, *daemonize=False*, *ssl=False*, *socket_host=None*, *socket_port=None*)

Bases: object

_join_daemon()

access_log = '/home/docs/checkouts/readthedocs.org/user_builds/cherrypy/envs/latest/li

config_file = '/home/docs/checkouts/readthedocs.org/user_builds/cherrypy/envs/latest/l

config_template = "[global]\nserver.socket_host: '%(host)s'\nserver.socket_port: %(p

error_log = '/home/docs/checkouts/readthedocs.org/user_builds/cherrypy/envs/latest/lib

get_pid()

join()
Wait for the process to exit.

pid_file = '/home/docs/checkouts/readthedocs.org/user_builds/cherrypy/envs/latest/lib/

start(*imports=None*)
Start cherryd in a subprocess.

write_conf(*extra=''*)

class cherrypy.test.helper.**CPWebCase**(*methodName='runTest'*)

Bases: cheroot.test.webtest.WebCase

classmethod _setup_server(*supervisor*, *conf*)

assertEqualDates(*dt1*, *dt2*, *seconds=None*)
Assert abs(dt1 - dt2) is within Y seconds.

assertErrorPage(*status*, *message=None*, *pattern=''*)
Compare the response body with a built in error page.

The function will optionally look for the regexp pattern, within the exception embedded in the error page.

available_servers = {'cpmodpy': <function get_cpmodpy_supervisor>, 'modfastcgi': <fu

base()

date_tolerance = 2

default_server = 'wsgi'

do_gc_test = False

exit()

getPage(*url*, **args*, ***kwargs*)
Open the url.

prefix()

scheme = 'http'

script_name = ''

classmethod setup_class()

skip(*msg='skipped '*)

classmethod teardown_class()

test_gc()

class cherrypy.test.helper.**LocalSupervisor**(***kwargs*)
Bases: *cherrypy.test.helper.Supervisor*

Base class for modeling/controlling servers which run in the same process.

When the server side runs in a different process, start/stop can dump all state between each test module easily. When the server side runs in the same process as the client, however, we have to do a bit more work to ensure config and mounted apps are reset between tests.

start(*modulename=None*)
Load and start the HTTP server.

stop()

sync_apps()
Tell the server about any apps which the setup functions mounted.

using_apache = False

using_wsgi = False

class cherrypy.test.helper.**LocalWSGISupervisor**(***kwargs*)
Bases: *cherrypy.test.helper.LocalSupervisor*

Server supervisor for the builtin WSGI server.

get_app(*app=None*)
Obtain a new (decorated) WSGI app to hook into the origin server.

httpserver_class = 'cherrypy._cpwsgi_server.CPWSGIServer'

sync_apps()
Hook a new WSGI app into the origin server.

using_apache = False

using_wsgi = True

class cherrypy.test.helper.**NativeServerSupervisor**(***kwargs*)
Bases: *cherrypy.test.helper.LocalSupervisor*

Server supervisor for the builtin HTTP server.

httpserver_class = 'cherrypy._cpnative_server.CPHTTPServer'

using_apache = False

using_wsgi = False

class cherrypy.test.helper.**Supervisor**(***kwargs*)
Bases: object

Base class for modeling and controlling servers during testing.

cherrypy.test.helper.**_test_method_sorter**(*_*, *x*, *y*)
Monkeypatch the test sorter to always run test_gc last in each suite.

cherrypy.test.helper.**get_cpmodpy_supervisor**(***options*)

cherrypy.test.helper.**get_modfastcgi_supervisor**(***options*)

cherrypy.test.helper.**get_modfcgid_supervisor**(***options*)

cherrypy.test.helper.**get_modpygw_supervisor**(***options*)

cherrypy.test.helper.**get_modwsgi_supervisor**(***options*)

cherrypy.test.helper.**get_wsgi_u_supervisor**(***options*)

cherrypy.test.helper.**log_to_stderr**(*msg*, *level*)

cherrypy.test.helper.**setup_client**()
Set up the WebCase classes to match the server's socket settings.

cherrypy.test.logtest module

logtest, a unittest.TestCase helper for testing log output.

class cherrypy.test.logtest.**LogCase**
Bases: object

unittest.TestCase mixin for testing log messages.

logfile: a filename for the desired log. Yes, I know modes are evil, but it makes the test functions so much cleaner to set this once.

lastmarker: the last marker in the log. This can be used to search for messages since the last marker.

markerPrefix: a string with which to prefix log markers. This should be unique enough from normal log output to use for marker identification.

_handleLogError(*msg*, *data*, *marker*, *pattern*)

_read_marked_region(*marker=None*)
Return lines from self.logfile in the marked region.

If marker is None, self.lastmarker is used. If the log hasn't been marked (using self.markLog), the entire log will be returned.

assertInLog(*line*, *marker=None*)
Fail if the given (partial) line is not in the log.

The log will be searched from the given marker to the next marker. If marker is None, self.lastmarker is used. If the log hasn't been marked (using self.markLog), the entire log will be searched.

assertLog(*sliceargs*, *lines*, *marker=None*)
Fail if log.readlines()[sliceargs] is not contained in 'lines'.

The log will be searched from the given marker to the next marker. If marker is None, self.lastmarker is used. If the log hasn't been marked (using self.markLog), the entire log will be searched.

assertNotInLog(*line*, *marker=None*)
Fail if the given (partial) line is in the log.

The log will be searched from the given marker to the next marker. If marker is None, self.lastmarker is used. If the log hasn't been marked (using self.markLog), the entire log will be searched.

assertValidUUIDv4(*marker=None*)
Fail if the given UUIDv4 is not valid.

The log will be searched from the given marker to the next marker. If marker is None, self.lastmarker is used. If the log hasn't been marked (using self.markLog), the entire log will be searched.

emptyLog()
Overwrite self.logfile with 0 bytes.

exit()

interactive = False

lastmarker = None

logfile = None

markLog(*key=None*)
Insert a marker line into the log and set self.lastmarker.

markerPrefix = b'test suite marker: '

cherrypy.test.logtest.**getchar**()

cherrypy.test.modfastcgi module

Wrapper for mod_fastcgi, for use as a CherryPy HTTP server when testing.

To autostart fastcgi, the "apache" executable or script must be on your system path, or you must override the global APACHE_PATH. On some platforms, "apache" may be called "apachectl", "apache2ctl", or "httpd"–create a symlink to them if needed.

You'll also need the WSGIServer from flup.servers. See http://projects.amor.org/misc/wiki/ModPythonGateway

KNOWN BUGS

1. **Apache processes Range headers automatically; CherryPy's truncated** output is then truncated again by Apache. See test_core.testRanges. This was worked around in http://www.cherrypy.org/changeset/1319.
2. **Apache does not allow custom HTTP methods like CONNECT as per the spec.** See test_core.testHTTPMethods.
3. Max request header and body settings do not work with Apache.
4. **Apache replaces status "reason phrases" automatically. For example,** CherryPy may set "304 Not modified" but Apache will write out "304 Not Modified" (capital "M").
5. Apache does not allow custom error codes as per the spec.
6. **Apache (or perhaps modpython, or modpython_gateway) unquotes %xx in the** Request-URI too early.
7. **mod_python will not read request bodies which use the "chunked"** transfer-coding (it passes REQUEST_CHUNKED_ERROR to ap_setup_client_block instead of REQUEST_CHUNKED_DECHUNK, see Apache2's http_protocol.c and mod_python's requestobject.c).
8. **Apache will output a "Content-Length: 0" response header even if there's** no response entity body. This isn't really a bug; it just differs from the CherryPy default.

class cherrypy.test.modfastcgi.**ModFCGISupervisor**(***kwargs*)
Bases: *cherrypy.test.helper.LocalWSGISupervisor*

httpserver_class = 'cherrypy.process.servers.FlupFCGIServer'

start(*modulename*)
Load and start the HTTP server.

start_apache()

stop()
Gracefully shutdown a server that is serving forever.

sync_apps()
Hook a new WSGI app into the origin server.

template = '\n# Apache2 server conf file for testing CherryPy with mod_fastcgi.\n# fum

using_apache = True

using_wsgi = True

cherrypy.test.modfastcgi.**erase_script_name**(*environ*, *start_response*)

cherrypy.test.modfastcgi.**read_process**(*cmd*, *args=''*)

cherrypy.test.modfcgid module

Wrapper for mod_fcgid, for use as a CherryPy HTTP server when testing.

To autostart fcgid, the "apache" executable or script must be on your system path, or you must override the global APACHE_PATH. On some platforms, "apache" may be called "apachectl", "apache2ctl", or "httpd"–create a symlink to them if needed.

You'll also need the WSGIServer from flup.servers. See http://projects.amor.org/misc/wiki/ModPythonGateway

KNOWN BUGS

1. **Apache processes Range headers automatically; CherryPy's truncated** output is then truncated again by Apache. See test_core.testRanges. This was worked around in http://www.cherrypy.org/changeset/1319.
2. **Apache does not allow custom HTTP methods like CONNECT as per the spec.** See test_core.testHTTPMethods.
3. Max request header and body settings do not work with Apache.
4. **Apache replaces status "reason phrases" automatically. For example,** CherryPy may set "304 Not modified" but Apache will write out "304 Not Modified" (capital "M").
5. Apache does not allow custom error codes as per the spec.
6. **Apache (or perhaps modpython, or modpython_gateway) unquotes %xx in the** Request-URI too early.
7. **mod_python will not read request bodies which use the "chunked"** transfer-coding (it passes REQUEST_CHUNKED_ERROR to ap_setup_client_block instead of REQUEST_CHUNKED_DECHUNK, see Apache2's http_protocol.c and mod_python's requestobject.c).
8. **Apache will output a "Content-Length: 0" response header even if there's** no response entity body. This isn't really a bug; it just differs from the CherryPy default.

class cherrypy.test.modfcgid.**ModFCGISupervisor**(***kwargs*)

Bases: *cherrypy.test.helper.LocalSupervisor*

start(*modulename*)
Load and start the HTTP server.

start_apache()

stop()
Gracefully shutdown a server that is serving forever.

sync_apps()
Tell the server about any apps which the setup functions mounted.

template = '\n# Apache2 server conf file for testing CherryPy with mod_fcgid.\n\nDocum

using_apache = True

using_wsgi = True

cherrypy.test.modfcgid.**read_process**(*cmd*, *args=''*)

cherrypy.test.modpy module

Wrapper for mod_python, for use as a CherryPy HTTP server when testing.

To autostart modpython, the "apache" executable or script must be on your system path, or you must override the global APACHE_PATH. On some platforms, "apache" may be called "apachectl" or "apache2ctl"– create a symlink to them if needed.

If you wish to test the WSGI interface instead of our _cpmodpy interface, you also need the 'modpython_gateway' module at: http://projects.amor.org/misc/wiki/ModPythonGateway

KNOWN BUGS

1. **Apache processes Range headers automatically; CherryPy's truncated** output is then truncated again by Apache. See test_core.testRanges. This was worked around in http://www.cherrypy.org/changeset/1319.
2. **Apache does not allow custom HTTP methods like CONNECT as per the spec.** See test_core.testHTTPMethods.
3. Max request header and body settings do not work with Apache.
4. **Apache replaces status "reason phrases" automatically. For example,** CherryPy may set "304 Not modified" but Apache will write out "304 Not Modified" (capital "M").
5. Apache does not allow custom error codes as per the spec.
6. **Apache (or perhaps modpython, or modpython_gateway) unquotes %xx in the** Request-URI too early.
7. **mod_python will not read request bodies which use the "chunked"** transfer-coding (it passes REQUEST_CHUNKED_ERROR to ap_setup_client_block instead of REQUEST_CHUNKED_DECHUNK, see Apache2's http_protocol.c and mod_python's requestobject.c).
8. **Apache will output a "Content-Length: 0" response header even if there's** no response entity body. This isn't really a bug; it just differs from the CherryPy default.

class cherrypy.test.modpy.**ModPythonSupervisor**(***kwargs*)

Bases: *cherrypy.test.helper.Supervisor*

start(*modulename*)

stop()

Gracefully shutdown a server that is serving forever.

template = None

using_apache = True

using_wsgi = False

cherrypy.test.modpy.**cpmodpysetup**(*req*)

cherrypy.test.modpy.**read_process**(*cmd*, *args=''*)

cherrypy.test.modpy.**wsgisetup**(*req*)

cherrypy.test.modwsgi module

Wrapper for mod_wsgi, for use as a CherryPy HTTP server.

To autostart modwsgi, the "apache" executable or script must be on your system path, or you must override the global APACHE_PATH. On some platforms, "apache" may be called "apachectl" or "apache2ctl"– create a symlink to them if needed.

KNOWN BUGS

1. **Apache processes Range headers automatically; CherryPy's truncated** output is then truncated again by Apache. See test_core.testRanges. This was worked around in http://www.cherrypy.org/changeset/1319.
2. **Apache does not allow custom HTTP methods like CONNECT as per the spec.** See test_core.testHTTPMethods.
3. Max request header and body settings do not work with Apache.
4. **Apache replaces status "reason phrases" automatically. For example,** CherryPy may set "304 Not modified" but Apache will write out "304 Not Modified" (capital "M").
5. Apache does not allow custom error codes as per the spec.
6. **Apache (or perhaps modpython, or modpython_gateway) unquotes %xx in the** Request-URI too early.
7. **mod_wsgi will not read request bodies which use the "chunked"** transfer-coding (it passes REQUEST_CHUNKED_ERROR to ap_setup_client_block instead of REQUEST_CHUNKED_DECHUNK, see Apache2's http_protocol.c and mod_python's requestobject.c).
8. **When responding with 204 No Content, mod_wsgi adds a Content-Length** header for you.
9. **When an error is raised, mod_wsgi has no facility for printing a** traceback as the response content (it's sent to the Apache log instead).
10. Startup and shutdown of Apache when running mod_wsgi seems slow.

class cherrypy.test.modwsgi.**ModWSGISupervisor**(***kwargs*)
Bases: *cherrypy.test.helper.Supervisor*

Server Controller for ModWSGI and CherryPy.

start(*modulename*)

stop()
Gracefully shutdown a server that is serving forever.

template = '\n# Apache2 server conf file for testing CherryPy with modpython_gateway.

using_apache = True

using_wsgi = True

cherrypy.test.modwsgi.**application**(*environ*, *start_response*)

cherrypy.test.modwsgi.**read_process**(*cmd*, *args=''*)

cherrypy.test.sessiondemo module

A session demonstration app.

class cherrypy.test.sessiondemo.**Root**
Bases: object

expire()

index()

page()

regen()

cherrypy.test.test_auth_basic module

class cherrypy.test.test_auth_basic.**BasicAuthTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testBasic()

testBasic2()

testBasic2_u()

testPublic()

cherrypy.test.test_auth_digest module

class cherrypy.test.test_auth_digest.**DigestAuthTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

_test_parametric_digest(*username*, *realm*)

static setup_server()

testPublic()

test_ascii_user()

test_unicode_user()

test_wrong_realm()

test_wrong_scheme()

cherrypy.test.test_auth_digest.**_fetch_users**()

cherrypy.test.test_bus module

Publish-subscribe bus tests.

cherrypy.test.test_bus.**bus**()
Return a wspbus instance.

cherrypy.test.test_bus.**listener**()
Return an instance of bus response tracker.

cherrypy.test.test_bus.**log_tracker**(*bus*)
Return an instance of bus log tracker.

cherrypy.test.test_bus.**test_block**(*bus*, *log_tracker*)
Test that bus block waits for exiting.

cherrypy.test.test_bus.**test_builtin_channels**(*bus*, *listener*)
Test that built-in channels trigger corresponding listeners.

cherrypy.test.test_bus.**test_custom_channels**(*bus*, *listener*)
Test that custom pub-sub channels work as built-in ones.

cherrypy.test.test_bus.**test_exit**(*bus*, *listener*, *log_tracker*)
Test that bus exit sequence is correct.

cherrypy.test.test_bus.**test_graceful**(*bus*, *listener*, *log_tracker*)
Test that bus graceful state triggers all listeners.

cherrypy.test.test_bus.**test_listener_errors**(*bus*, *listener*)
Test that unhandled exceptions raise channel failures.

cherrypy.test.test_bus.**test_log**(*bus*, *log_tracker*)
Test that bus messages and errors are logged.

cherrypy.test.test_bus.**test_start**(*bus*, *listener*, *log_tracker*)
Test that bus start sequence calls all listeners.

cherrypy.test.test_bus.**test_start_with_callback**(*bus*)
Test that callback fires on bus start.

cherrypy.test.test_bus.**test_stop**(*bus*, *listener*, *log_tracker*)
Test that bus stop sequence calls all listeners.

cherrypy.test.test_bus.**test_wait**(*bus*)
Test that bus wait awaits for states.

cherrypy.test.test_bus.**test_wait_publishes_periodically**(*bus*)
Test that wait publishes each tick.

cherrypy.test.test_caching module

class cherrypy.test.test_caching.**CacheTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

_assert_resp_len_and_enc_for_gzip(*uri*)
Test that after querying gzipped content it's remains valid in cache and available non-gzipped as well.

static setup_server()

testCaching()

testExpiresTool()

testGzipStaticCache()
Test that cache and gzip tools play well together when both enabled.

Ref GitHub issue #1190.

testLastModified()

testVaryHeader()

test_antistampede()

test_cache_control()

cherrypy.test.test_config module

Tests for the CherryPy configuration system.

class cherrypy.test.test_config.**CallablesInConfigTest**(*methodName='runTest'*)
Bases: unittest.case.TestCase

static setup_server()

test_call_with_kwargs()

test_call_with_literal_dict()

class cherrypy.test.test_config.**ConfigTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testConfig()

testCustomNamespaces()

testHandlerToolConfigOverride()

testRespNamespaces()

testUnrepr()

test_request_body_namespace()

cherrypy.test.test_config.**StringIOFromNative**(*x*)

class cherrypy.test.test_config.**VariableSubstitutionTests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

static setup_server()

test_config()

cherrypy.test.test_config.**setup_server**()

cherrypy.test.test_config_server module

Tests for the CherryPy configuration system.

class cherrypy.test.test_config_server.**ServerConfigTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

PORT = 9876

static setup_server()

testAdditionalServers()

testBasicConfig()

testMaxRequestSize()

testMaxRequestSizePerHandler()

cherrypy.test.test_conn module

Tests for TCP connection handling, including proper and timely close.

class cherrypy.test.test_conn.**BadRequestTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_No_CRLF()

class cherrypy.test.test_conn.**ConnectionCloseTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

_streaming(*set_cl*)

static setup_server()

test_HTTP10_KeepAlive()

test_HTTP11()

test_Streaming_no_len()

test_Streaming_with_len()

class cherrypy.test.test_conn.**ConnectionTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_598()

test_Chunked_Encoding()

test_Content_Length_in()

test_Content_Length_out_postheaders()

test_Content_Length_out_preheaders()

test_No_Message_Body()

test_readall_or_close()

class cherrypy.test.test_conn.**LimitedRequestQueueTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_queue_full()

class cherrypy.test.test_conn.**PipelineTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_100_Continue()

test_HTTP11_Timeout()

test_HTTP11_Timeout_after_request()

test_HTTP11_pipelining()

cherrypy.test.test_conn.**setup_server**()

cherrypy.test.test_conn.**setup_upload_server**()

cherrypy.test.test_conn.**socket_reset_errors = [104, 'Remote end closed connection without r**
reset error numbers available on this platform

cherrypy.test.test_core module

Basic tests for the CherryPy core: request handling.

class cherrypy.test.test_core.**CoreRequestHandlingTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

skip_if_bad_cookies()
cookies module fails to reject invalid cookies https://github.com/cherrypy/cherrypy/issues/1405

testCookies()

testDefaultContentType()

testFavicon()

testFlatten()

testRanges()

testRedirect()

testSlashes()

testStatus()

test_InternalRedirect()

test_cherrypy_url()

test_expose_decorator()

test_multiple_headers()

test_on_end_resource_status()

test_redirect_with_unicode()
A redirect to a URL with Unicode should return a Location header containing that Unicode URL.

test_redirect_with_xss()
A redirect to a URL with HTML injected should result in page contents escaped.

class cherrypy.test.test_core.**ErrorTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_contextmanager()

test_start_response_error()

class cherrypy.test.test_core.**TestBinding**
Bases: object

test_bind_ephemeral_port()
A server configured to bind to port 0 will bind to an ephemeral port and indicate that port number on startup.

cherrypy.test.test_dynamicobjectmapping module

class cherrypy.test.test_dynamicobjectmapping.**DynamicObjectMappingTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testMethodDispatch()

testObjectMapping()

testVpathDispatch()

cherrypy.test.test_dynamicobjectmapping.**setup_server**()

cherrypy.test.test_encoding module

class cherrypy.test.test_encoding.**EncodingTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testEncoding()

testGzip()

test_BytesHeaders()

test_UnicodeHeaders()

test_decode_tool()

test_multipart_decoding()

test_multipart_decoding_bigger_maxrambytes()
Decoding of a multipart entity should also pass when the entity is bigger than maxrambytes. See ticket #1352.

test_multipart_decoding_no_charset()

test_multipart_decoding_no_successful_charset()

test_nontext()

test_query_string_decoding()

test_urlencoded_decoding()

cherrypy.test.test_etags module

class cherrypy.test.test_etags.**ETagTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_errors()

test_etags()

test_unicode_body()

cherrypy.test.test_http module

Tests for managing HTTP issues (malformed requests, etc).

class cherrypy.test.test_http.**HTTPTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

make_connection()

static setup_server()

test_garbage_in()

test_http_over_https()

test_malformed_header()

test_malformed_request_line()

test_no_content_length()

test_post_filename_with_special_characters()
Testing that we can handle filenames with special characters.

This was reported as a bug in:

- https://github.com/cherrypy/cherrypy/issues/1146/
- https://github.com/cherrypy/cherrypy/issues/1397/
- https://github.com/cherrypy/cherrypy/issues/1694/

test_post_multipart()

test_request_line_split_issue_1220()

cherrypy.test.test_http.**encode_filename**(*filename*)
Given a filename to be used in a multipart/form-data, encode the name. Return the key and encoded filename.

cherrypy.test.test_http.**encode_multipart_formdata**(*files*)
Return (content_type, body) ready for httplib.HTTP instance.

files: a sequence of (name, filename, value) tuples for multipart uploads. filename can be a string or a tuple ('filename string', 'encoding')

cherrypy.test.test_http.**is_ascii**(*text*)
Return True if the text encodes as ascii.

cherrypy.test.test_httputil module

Test helpers from `cherrypy.lib.httputil` module.

`cherrypy.test.test_httputil.`**`test_invalid_status`**(*status_code*, *error_msg*)
Check that invalid status cause certain errors.

`cherrypy.test.test_httputil.`**`test_urljoin`**(*script_name*, *path_info*, *expected_url*)
Test all slash+atom combinations for SCRIPT_NAME and PATH_INFO.

`cherrypy.test.test_httputil.`**`test_valid_status`**(*status*, *expected_status*)
Check valid int, string and http.client-constants statuses processing.

cherrypy.test.test_iterator module

class cherrypy.test.test_iterator.**IteratorBase**
Bases: `object`

created = 0

datachunk = 'butternut squashbutternut squashbutternut squashbutternut squashbutternut

classmethod decr()

classmethod incr()

class cherrypy.test.test_iterator.**IteratorTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

_test_iterator()

static setup_server()

test_iterator()

class cherrypy.test.test_iterator.**OurClosableIterator**
Bases: *cherrypy.test.test_iterator.OurIterator*

close()

class cherrypy.test.test_iterator.**OurGenerator**
Bases: *cherrypy.test.test_iterator.IteratorBase*

class cherrypy.test.test_iterator.**OurIterator**
Bases: *cherrypy.test.test_iterator.IteratorBase*

closed_off = False

count = 0

decrement()

increment()

next()

started = False

class cherrypy.test.test_iterator.**OurNotClosableIterator**
Bases: *cherrypy.test.test_iterator.OurIterator*

close(*somearg*)

class cherrypy.test.test_iterator.**OurUnclosableIterator**
Bases: *cherrypy.test.test_iterator.OurIterator*

close = 'close'

cherrypy.test.test_json module

class cherrypy.test.test_json.**JsonTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_cached()

test_json_input()

test_json_output()

cherrypy.test.test_logging module

Basic tests for the CherryPy core: request handling.

cherrypy.test.test_logging.**access_log_file**(*tmp_path_factory*)

cherrypy.test.test_logging.**configure_server**(*access_log_file*, *error_log_file*)

cherrypy.test.test_logging.**error_log_file**(*tmp_path_factory*)

cherrypy.test.test_logging.**log_tracker**(*access_log_file*)

cherrypy.test.test_logging.**server**(*configure_server*)

cherrypy.test.test_logging.**shutdown_server**()

cherrypy.test.test_logging.**test_UUIDv4_parameter_log_format**(*log_tracker*, *monkeypatch*, *server*)
Test rendering of UUID4 within access log.

cherrypy.test.test_logging.**test_custom_log_format**(*log_tracker*, *monkeypatch*, *server*)
Test a customized access_log_format string, which is a feature of _cplogging.LogManager.access().

cherrypy.test.test_logging.**test_escaped_output**(*log_tracker*, *server*)

cherrypy.test.test_logging.**test_normal_return**(*log_tracker*, *server*)

cherrypy.test.test_logging.**test_normal_yield**(*log_tracker*, *server*)

cherrypy.test.test_logging.**test_timez_log_format**(*log_tracker*, *monkeypatch*, *server*)
Test a customized access_log_format string, which is a feature of _cplogging.LogManager.access().

cherrypy.test.test_logging.**test_tracebacks**(*server*, *caplog*)

cherrypy.test.test_mime module

Tests for various MIME issues, including the safe_multipart Tool.

class cherrypy.test.test_mime.**MultipartTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_multipart()

test_multipart_form_data()

class cherrypy.test.test_mime.**SafeMultipartHandlingTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_Flash_Upload()

cherrypy.test.test_mime.**setup_server**()

cherrypy.test.test_misc_tools module

class cherrypy.test.test_misc_tools.**AcceptTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_Accept_Tool()

test_accept_selection()

class cherrypy.test.test_misc_tools.**AutoVaryTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testAutoVary()

class cherrypy.test.test_misc_tools.**RefererTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testReferer()

class cherrypy.test.test_misc_tools.**ResponseHeadersTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testResponseHeaders()

testResponseHeadersDecorator()

cherrypy.test.test_misc_tools.**setup_server**()

cherrypy.test.test_native module

Test the native server.

cherrypy.test.test_native.**cp_native_server**(*request*)
A native server.

cherrypy.test.test_native.**test_basic_request**(*cp_native_server*)
A request to a native server should succeed.

cherrypy.test.test_objectmapping module

class cherrypy.test.test_objectmapping.**ObjectMappingTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testExpose()

testKeywords()

testMethodDispatch()

testObjectMapping()

testPositionalParams()

testTreeMounting()

test_redir_using_url()

test_translate()

cherrypy.test.test_params module

class cherrypy.test.test_params.**ParamsTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_error()

test_pass()

test_syntax()

cherrypy.test.test_plugins module

class cherrypy.test.test_plugins.**TestAutoreloader**
Bases: object

test_file_for_file_module_when_None()
No error when module.__file__ is None.

cherrypy.test.test_proxy module

class cherrypy.test.test_proxy.**ProxyTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testProxy()

test_no_base_port_in_host()
If no base is indicated, and the host header is used to resolve the base, it should rely on the host header for the port also.

cherrypy.test.test_refleaks module

Tests for refleaks.

class cherrypy.test.test_refleaks.**ReferenceTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_threadlocal_garbage()

cherrypy.test.test_request_obj module

Basic tests for the cherrypy.Request object.

class cherrypy.test.test_request_obj.**RequestObjectTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testAbsoluteURIPathInfo()

testEmptyThreadlocals()

testErrorHandling()

testExpect()

testHeaderElements()

testParamErrors()

testParams()

testRelativeURIPathInfo()

test_CONNECT_method()

test_CONNECT_method_invalid_authority()

test_basic_HTTPMethods()

test_encoded_headers()

test_header_presence()

test_per_request_uuid4()

test_repeated_headers()

test_scheme()

cherrypy.test.test_routes module

Test Routes dispatcher.

class cherrypy.test.test_routes.**RoutesDispatchTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

Routes dispatcher test suite.

static setup_server()
Set up cherrypy test instance.

test_Routes_Dispatch()
Check that routes package based URI dispatching works correctly.

cherrypy.test.test_session module

class cherrypy.test.test_session.**MemcachedSessionTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

pytestmark = [Mark(name='usefixtures', args=('memcached_configured',), kwargs={}), Mar

static setup_server()

test_0_Session()

test_1_Concurrency()

test_3_Redirect()

test_5_Error_paths()

class cherrypy.test.test_session.**SessionTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

_test_Concurrency()

static setup_server()

classmethod teardown_class()
Clean up sessions.

test_0_Session()

test_1_Ram_Concurrency()

test_2_File_Concurrency()

test_3_Redirect()

test_4_File_deletion()

test_5_Error_paths()

test_6_regenerate()

test_7_session_cookies()

test_8_Ram_Cleanup()

cherrypy.test.test_session.**http_methods_allowed**(*methods=['GET', 'HEAD']*)

cherrypy.test.test_session.**is_memcached_present**()

cherrypy.test.test_session.**memcached_client_present**()

cherrypy.test.test_session.**memcached_configured**(*memcached_instance*, *monkeypatch*, *memcached_client_present*)

cherrypy.test.test_session.**memcached_instance**(*request*, *watcher_getter*, *memcached_server_present*)
Start up an instance of memcached.

cherrypy.test.test_session.**memcached_server_present**()

cherrypy.test.test_session.**setup_server**()

cherrypy.test.test_sessionauthenticate module

class cherrypy.test.test_sessionauthenticate.**SessionAuthenticateTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testSessionAuthenticate()

cherrypy.test.test_states module

class cherrypy.test.test_states.**Dependency**(*bus*)
Bases: object

graceful()

start()

startthread(*thread_id*)

stop()

stopthread(*thread_id*)

subscribe()

class cherrypy.test.test_states.**PluginTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

test_daemonize()

class cherrypy.test.test_states.**ServerStateTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

setUp()
Hook method for setting up the test fixture before exercising it.

static setup_server()

test_0_NormalStateFlow()

test_1_Restart()

test_2_KeyboardInterrupt()

test_4_Autoreload()

test_5_Start_Error()

class cherrypy.test.test_states.**SignalHandlingTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

_require_signal_and_kill(*signal_name*)

test_SIGHUP_daemonized()

test_SIGHUP_tty()

test_SIGTERM()
SIGTERM should shut down the server whether daemonized or not.

test_signal_handler_unsubscribe()

cherrypy.test.test_states.**setup_server**()

cherrypy.test.test_states.**test_safe_wait_INADDR_ANY**()
Wait on INADDR_ANY should not raise IOError

In cases where the loopback interface does not exist, CherryPy cannot effectively determine if a port binding to INADDR_ANY was effected. In this situation, CherryPy should assume that it failed to detect the binding (not that the binding failed) and only warn that it could not verify it.

cherrypy.test.test_static module

class cherrypy.test.test_static.**StaticTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

files_to_remove = []

static setup_server()

classmethod teardown_class()

test_755_vhost()

test_config_errors()

test_error_page_with_serve_file()

test_fallthrough()

test_file_stream()

test_file_stream_deadlock()

test_index()

test_modif()

test_null_bytes()

test_security()

test_serve_bytesio()

test_serve_fileobj()

test_static()

test_static_longpath()
Test serving of a file in subdir of a Windows long-path staticdir.

test_unicode()

classmethod unicode_file()

cherrypy.test.test_static.**_check_unicode_filesystem**(*tmpdir*)

cherrypy.test.test_static.**ensure_unicode_filesystem**()
TODO: replace with simply pytest fixtures once webtest.TestCase no longer implies unittest.

cherrypy.test.test_static.**error_page_404**(*status*, *message*, *traceback*, *version*)

cherrypy.test.test_static.**unicode_filesystem**(*tmpdir*)

cherrypy.test.test_tools module

Test the various means of instantiating and invoking tools.

class cherrypy.test.test_tools.**SessionAuthTest**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_login_screen_returns_bytes()
login_screen must return bytes even if unicode parameters are passed. Issue 1132 revealed that login_screen would return unicode if the username and password were unicode.

class cherrypy.test.test_tools.**TestHooks**
Bases: object

test_priorities()
Hooks should sort by priority order.

class cherrypy.test.test_tools.**ToolTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testBareHooks()

testCombinedTools()

testDecorator()

testEndRequestOnDrop()

testGuaranteedHooks()

testHandlerWrapperTool()

testHookErrors()

testToolWithConfig()

testWarnToolOn()

cherrypy.test.test_tutorials module

class cherrypy.test.test_tutorials.**TutorialTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static load_module(*name*)
Import or reload tutorial module as needed.

classmethod setup_server()
Mount something so the engine starts.

classmethod setup_tutorial(*name*, *root_name*, *config={}*)

test01HelloWorld()

test02ExposeMethods()

test03GetAndPost()

test04ComplexSite()

test05DerivedObjects()

test06DefaultMethod()

test07Sessions()

test08GeneratorsAndYield()

test09Files()

test10HTTPErrors()

cherrypy.test.test_virtualhost module

class cherrypy.test.test_virtualhost.**VirtualHostTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testVirtualHost()

test_VHost_plus_Static()

cherrypy.test.test_wsgi_ns module

class cherrypy.test.test_wsgi_ns.**WSGI_Namespace_Test**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_pipeline()

cherrypy.test.test_wsgi_unix_socket module

class cherrypy.test.test_wsgi_unix_socket.**USocketHTTPConnection**(*path*)
Bases: http.client.HTTPConnection

HTTPConnection over a unix socket.

connect()
Override the connect method and assign a unix socket as a transport.

class cherrypy.test.test_wsgi_unix_socket.**WSGI_UnixSocket_Test**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

Test basic behavior on a cherrypy wsgi server listening on a unix socket.

It exercises the config option `server.socket_file`.

HTTP_CONN = <cherrypy.test.test_wsgi_unix_socket.USocketHTTPConnection object>

pytestmark = [Mark(name='skipif', args=("sys.platform == 'win32'",), kwargs={})]

static setup_server()

tearDown()
Hook method for deconstructing the test fixture after testing it.

test_internal_error()

test_not_found()

test_simple_request()

cherrypy.test.test_wsgi_unix_socket.**usocket_path**()

cherrypy.test.test_wsgi_vhost module

class cherrypy.test.test_wsgi_vhost.**WSGI_VirtualHost_Test**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_welcome()

cherrypy.test.test_wsgiapps module

class cherrypy.test.test_wsgiapps.**WSGIGraftTests**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

test_01_standard_app()

test_04_pure_wsgi()

test_05_wrapped_cp_app()

test_06_empty_string_app()

wsgi_output = 'Hello, world!\nThis is a wsgi app running within CherryPy!'

cherrypy.test.test_xmlrpc module

class cherrypy.test.test_xmlrpc.**XmlRpcTest**(*methodName='runTest'*)
Bases: *cherrypy.test.helper.CPWebCase*

static setup_server()

testXmlRpc()

cherrypy.test.test_xmlrpc.**setup_server**()

cherrypy.test.webtest module

Module contents

Regression test suite for CherryPy.

cherrypy.test.**newexit**()

cherrypy.test.**setup**()

cherrypy.test.**teardown**()

cherrypy.tutorial package

Submodules

cherrypy.tutorial.tut01_helloworld module

Tutorial - Hello World

The most basic (working) CherryPy application possible.

class cherrypy.tutorial.tut01_helloworld.**HelloWorld**
Bases: object

Sample request handler class.

index()

cherrypy.tutorial.tut02_expose_methods module

Tutorial - Multiple methods

This tutorial shows you how to link to other methods of your request handler.

class cherrypy.tutorial.tut02_expose_methods.**HelloWorld**
Bases: object

index()

show_msg()

cherrypy.tutorial.tut03_get_and_post module

Tutorial - Passing variables

This tutorial shows you how to pass GET/POST variables to methods.

class cherrypy.tutorial.tut03_get_and_post.**WelcomePage**
Bases: object

greetUser(*name=None*)

index()

cherrypy.tutorial.tut04_complex_site module

Tutorial - Multiple objects

This tutorial shows you how to create a site structure through multiple possibly nested request handler objects.

class cherrypy.tutorial.tut04_complex_site.**ExtraLinksPage**
Bases: object

index()

class cherrypy.tutorial.tut04_complex_site.**HomePage**
Bases: object

index()

class cherrypy.tutorial.tut04_complex_site.**JokePage**
Bases: object

index()

class cherrypy.tutorial.tut04_complex_site.**LinksPage**
Bases: object

index()

cherrypy.tutorial.tut05_derived_objects module

Tutorial - Object inheritance

You are free to derive your request handler classes from any base class you wish. In most real-world applications, you will probably want to create a central base class used for all your pages, which takes care of things like printing a common page header and footer.

class cherrypy.tutorial.tut05_derived_objects.**AnotherPage**
Bases: *cherrypy.tutorial.tut05_derived_objects.Page*

index()

title = 'Another Page'

class cherrypy.tutorial.tut05_derived_objects.**HomePage**
Bases: *cherrypy.tutorial.tut05_derived_objects.Page*

index()

title = 'Tutorial 5'

class cherrypy.tutorial.tut05_derived_objects.**Page**
Bases: object

footer()

header()

title = 'Untitled Page'

cherrypy.tutorial.tut06_default_method module

Tutorial - The default method

Request handler objects can implement a method called “default” that is called when no other suitable method/object could be found. Essentially, if CherryPy2 can’t find a matching request handler object for the given request URI, it will use the default method of the object located deepest on the URI path.

Using this mechanism you can easily simulate virtual URI structures by parsing the extra URI string, which you can access through cherrypy.request.virtualPath.

The application in this tutorial simulates an URI structure looking like /users/<username>. Since the <username> bit will not be found (as there are no matching methods), it is handled by the default method.

class cherrypy.tutorial.tut06_default_method.**UsersPage**
Bases: object

default(*user*)

index()

cherrypy.tutorial.tut07_sessions module

Tutorial - Sessions

Storing session data in CherryPy applications is very easy: cherrypy provides a dictionary called "session" that represents the session data for the current user. If you use RAM based sessions, you can store any kind of object into that dictionary; otherwise, you are limited to objects that can be pickled.

class cherrypy.tutorial.tut07_sessions.**HitCounter**
Bases: object

_cp_config = {'tools.sessions.on': True}

index()

cherrypy.tutorial.tut08_generators_and_yield module

Bonus Tutorial: Using generators to return result bodies

Instead of returning a complete result string, you can use the yield statement to return one result part after another. This may be convenient in situations where using a template package like CherryPy or Cheetah would be overkill, and messy string concatenation too uncool. ;-)

class cherrypy.tutorial.tut08_generators_and_yield.**GeneratorDemo**
Bases: object

footer()

header()

index()

cherrypy.tutorial.tut09_files module

Tutorial: File upload and download

Uploads

When a client uploads a file to a CherryPy application, it's placed on disk immediately. CherryPy will pass it to your exposed method as an argument (see "myFile" below); that arg will have a "file" attribute, which is a handle to the temporary uploaded file. If you wish to permanently save the file, you need to read() from myFile.file and write() somewhere else.

Note the use of 'enctype="multipart/form-data"' and 'input type="file"' in the HTML which the client uses to upload the file.

Downloads

If you wish to send a file to the client, you have two options: First, you can simply return a file-like object from your page handler. CherryPy will read the file and serve it as the content (HTTP body) of the response. However, that doesn't tell the client that the response is a file to be saved, rather than displayed. Use cherrypy.lib.static.serve_file for that; it takes four arguments:

serve_file(path, content_type=None, disposition=None, name=None)

Set "name" to the filename that you expect clients to use when they save your file. Note that the "name" argument is ignored if you don't also provide a "disposition" (usually "attachement"). You can manually set "content_type", but be aware that if you also use the encoding tool, it may choke if the file extension is not recognized as belonging to a known Content-Type. Setting the content_type to "application/x-download" works in most cases, and should prompt the user with an Open/Save dialog in popular browsers.

class cherrypy.tutorial.tut09_files.**FileDemo**
Bases: object

download()

index()

upload(*myFile*)

cherrypy.tutorial.tut10_http_errors module

Tutorial: HTTP errors

HTTPError is used to return an error response to the client. CherryPy has lots of options regarding how such errors are logged, displayed, and formatted.

class cherrypy.tutorial.tut10_http_errors.**HTTPErrorDemo**
Bases: object

_cp_config = {'error_page.403': '/home/docs/checkouts/readthedocs.org/user_builds/che

error(*code*)

index()

messageArg()

toggleTracebacks()

Module contents

15.1.2 Submodules

cherrypy.__main__ module

CherryPy'd cherryd daemon runner.

cherrypy._cpchecker module

Checker for CherryPy sites and mounted apps.

class cherrypy._cpchecker.**Checker**
Bases: object

A checker for CherryPy sites and their mounted applications.

When this object is called at engine startup, it executes each of its own methods whose names start with check_. If you wish to disable selected checks, simply add a line in your global config which sets the appropriate method to False:

```
[global]
checker.check_skipped_app_config = False
```

You may also dynamically add or replace check_* methods in this way.

_compat(*config*)
Process config and warn on each obsolete or deprecated entry.

_known_ns(*app*)

_known_types(*config*)

_populate_known_types()

check_app_config_brackets()
Check for App config with extraneous brackets in section names.

check_app_config_entries_dont_start_with_script_name()
Check for App config with sections that repeat script_name.

check_compatibility()
Process config and warn on each obsolete or deprecated entry.

check_config_namespaces()
Process config and warn on each unknown config namespace.

check_config_types()
Assert that config values are of the same type as default values.

check_localhost()
Warn if any socket_host is 'localhost'. See #711.

check_site_config_entries_in_app_config()
Check for mounted Applications that have site-scoped config.

check_skipped_app_config()
Check for mounted Applications that have no config.

check_static_paths()
Check Application config for incorrect static paths.

deprecated = {}

extra_config_namespaces = []

formatwarning(*message*, *category*, *filename*, *lineno*, *line=None*)
Format a warning.

global_config_contained_paths = False

known_config_types = {'engine.__class__': <class 'type'>, 'engine.__dict__': <class

obsolete = {'log_access_file': 'log.access_file', 'log_config_options': None, 'log_f

on = True
If True (the default), run all checks; if False, turn off all checks.

cherrypy._cpcompat module

Compatibility code for using CherryPy with various versions of Python.

To retain compatibility with older Python versions, this module provides a useful abstraction over the differences between Python versions, sometimes by preferring a newer idiom, sometimes an older one, and sometimes a custom one.

In particular, Python 2 uses str and '' for byte strings, while Python 3 uses str and '' for unicode strings. We will call each of these the 'native string' type for each version. Because of this major difference, this module provides two functions: 'ntob', which translates native strings (of type 'str') into byte strings regardless of Python version, and 'ntou', which translates native strings to unicode strings.

Try not to use the compatibility functions 'ntob', 'ntou', 'tonative'. They were created with Python 2.3-2.5 compatibility in mind. Instead, use unicode literals (from __future__) and bytes literals and their .encode/.decode methods as needed.

cherrypy._cpcompat.**assert_native**(*n*)

cherrypy._cpcompat.**ntob**(*n*, *encoding='ISO-8859-1'*)
Return the given native string as a byte string in the given encoding.

cherrypy._cpcompat.**ntou**(*n*, *encoding='ISO-8859-1'*)
Return the given native string as a unicode string with the given encoding.

cherrypy._cpcompat.**tonative**(*n*, *encoding='ISO-8859-1'*)
Return the given string as a native string in the given encoding.

cherrypy._cpconfig module

Configuration system for CherryPy.

Configuration in CherryPy is implemented via dictionaries. Keys are strings which name the mapped value, which may be of any type.

Architecture

CherryPy Requests are part of an Application, which runs in a global context, and configuration data may apply to any of those three scopes:

Global Configuration entries which apply everywhere are stored in cherrypy.config.

Application Entries which apply to each mounted application are stored on the Application object itself, as 'app.config'. This is a two-level dict where each key is a path, or "relative URL" (for example, "/" or "/path/to/my/page"), and each value is a config dict. Usually, this data is provided in the call to tree.mount(root(), config=conf), although you may also use app.merge(conf).

Request Each Request object possesses a single 'Request.config' dict. Early in the request process, this dict is populated by merging global config entries, Application entries (whose path equals or is a parent of Request.path_info), and any config acquired while looking up the page handler (see next).

Declaration

Configuration data may be supplied as a Python dictionary, as a filename, or as an open file object. When you supply a filename or file, CherryPy uses Python's builtin ConfigParser; you declare Application config by writing each path as a section header:

```
[/path/to/my/page]
request.stream = True
```

To declare global configuration entries, place them in a [global] section.

You may also declare config entries directly on the classes and methods (page handlers) that make up your CherryPy application via the `_cp_config` attribute, set with the `cherrypy.config` decorator. For example:

```
@cherrypy.config(**{'tools.gzip.on': True})
class Demo:

    @cherrypy.expose
    @cherrypy.config(**{'request.show_tracebacks': False})
    def index(self):
        return "Hello world"
```

Note: This behavior is only guaranteed for the default dispatcher. Other dispatchers may have different restrictions on where you can attach config attributes.

Namespaces

Configuration keys are separated into namespaces by the first "." in the key. Current namespaces:

engine Controls the 'application engine', including autoreload. These can only be declared in the global config.

tree Grafts cherrypy.Application objects onto cherrypy.tree. These can only be declared in the global config.

hooks Declares additional request-processing functions.

log Configures the logging for each application. These can only be declared in the global or / config.

request Adds attributes to each Request.

response Adds attributes to each Response.

server Controls the default HTTP server via cherrypy.server. These can only be declared in the global config.

tools Runs and configures additional request-processing packages.

wsgi Adds WSGI middleware to an Application's "pipeline". These can only be declared in the app's root config ("/").

checker Controls the 'checker', which looks for common errors in app state (including config) when the engine starts. Global config only.

The only key that does not exist in a namespace is the "environment" entry. This special entry 'imports' other config entries from a template stored in cherrypy._cpconfig.environments[environment]. It only applies to the global config, and only when you use cherrypy.config.update.

You can define your own namespaces to be called at the Global, Application, or Request level, by adding a named handler to cherrypy.config.namespaces, app.namespaces, or app.request_class.namespaces. The name can be any string, and the handler must be either a callable or a (Python 2.5 style) context manager.

class cherrypy._cpconfig.**Config**(*file=None*, ***kwargs*)
Bases: *cherrypy.lib.reprconf.Config*

The 'global' configuration data for the entire CherryPy process.

_apply(*config*)
Update self from a dict.

environments = {'embedded': {'checker.on': False, 'engine.SIGHUP': None, 'engine.SIG

update(*config*)
Update self from a dict, file or filename.

class cherrypy._cpconfig.**_Vars**(*target*)
Bases: object

Adapter allowing setting a default attribute on a function or class.

setdefault(*key*, *default*)

cherrypy._cpconfig.**_engine_namespace_handler**(*k*, *v*)
Config handler for the "engine" namespace.

cherrypy._cpconfig.**_if_filename_register_autoreload**(*ob*)
Register for autoreload if ob is a string (presumed filename).

cherrypy._cpconfig.**_server_namespace_handler**(*k*, *v*)
Config handler for the "server" namespace.

cherrypy._cpconfig.**_tree_namespace_handler**(*k*, *v*)
Namespace handler for the 'tree' config namespace.

cherrypy._cpconfig.**merge**(*base*, *other*)
Merge one app config (from a dict, file, or filename) into another.

If the given config is a filename, it will be appended to the list of files to monitor for "autoreload" changes.

cherrypy._cpdispatch module

CherryPy dispatchers.

A 'dispatcher' is the object which looks up the 'page handler' callable and collects config for the current request based on the path_info, other request attributes, and the application architecture. The core calls the dispatcher as early as possible, passing it a 'path_info' argument.

The default dispatcher discovers the page handler by matching path_info to a hierarchical arrangement of objects, starting at request.app.root.

class cherrypy._cpdispatch.**Dispatcher**(*dispatch_method_name=None*, *translate={33: 95, 34: 95, 35: 95, 36: 95, 37: 95, 38: 95, 39: 95, 40: 95, 41: 95, 42: 95, 43: 95, 44: 95, 45: 95, 46: 95, 47: 95, 58: 95, 59: 95, 60: 95, 61: 95, 62: 95, 63: 95, 64: 95, 91: 95, 92: 95, 93: 95, 94: 95, 95: 95, 96: 95, 123: 95, 124: 95, 125: 95, 126: 95}*)
Bases: object

CherryPy Dispatcher which walks a tree of objects to find a handler.

The tree is rooted at cherrypy.request.app.root, and each hierarchical component in the path_info argument is matched to a corresponding nested attribute of the root object. Matching handlers must have an 'exposed' attribute which evaluates to True. The special method name "index" matches a URI which ends in a slash ("/").

The special method name "default" may match a portion of the path_info (but only when no longer substring of the path_info matches some other object).

This is the default, built-in dispatcher for CherryPy.

dispatch_method_name = '_cp_dispatch'
The name of the dispatch method that nodes may optionally implement to provide their own dynamic dispatch algorithm.

find_handler(*path*)
Return the appropriate page handler, plus any virtual path.

This will return two objects. The first will be a callable, which can be used to generate page output. Any parameters from the query string or request body will be sent to that callable as keyword arguments.

The callable is found by traversing the application's tree, starting from cherrypy.request.app.root, and matching path components to successive objects in the tree. For example, the URL "/path/to/handler" might return root.path.to.handler.

The second object returned will be a list of names which are 'virtual path' components: parts of the URL which are dynamic, and were not used when looking up the handler. These virtual path components are passed to the handler as positional arguments.

class cherrypy._cpdispatch.**LateParamPageHandler**(*callable*, **args*, ***kwargs*)
Bases: *cherrypy._cpdispatch.PageHandler*

When passing cherrypy.request.params to the page handler, we do not want to capture that dict too early; we want to give tools like the decoding tool a chance to modify the params dict in-between the lookup of the handler and the actual calling of the handler. This subclass takes that into account, and allows request.params to be 'bound late' (it's more complicated than that, but that's the effect).

property kwargs
Page handler kwargs (with cherrypy.request.params copied in).

class cherrypy._cpdispatch.**MethodDispatcher**(*dispatch_method_name=None*, *translate={33: 95, 34: 95, 35: 95, 36: 95, 37: 95, 38: 95, 39: 95, 40: 95, 41: 95, 42: 95, 43: 95, 44: 95, 45: 95, 46: 95, 47: 95, 58: 95, 59: 95, 60: 95, 61: 95, 62: 95, 63: 95, 64: 95, 91: 95, 92: 95, 93: 95, 94: 95, 95: 95, 96: 95, 123: 95, 124: 95, 125: 95, 126: 95}*)
Bases: *cherrypy._cpdispatch.Dispatcher*

Additional dispatch based on cherrypy.request.method.upper().

Methods named GET, POST, etc will be called on an exposed class. The method names must be all caps; the appropriate Allow header will be output showing all capitalized method names as allowable HTTP verbs.

Note that the containing class must be exposed, not the methods.

class cherrypy._cpdispatch.**PageHandler**(*callable*, **args*, ***kwargs*)
Bases: object

Callable which sets response.body.

property args
The ordered args should be accessible from post dispatch hooks.

property kwargs
The named kwargs should be accessible from post dispatch hooks.

class cherrypy._cpdispatch.**RoutesDispatcher**(*full_result=False*, ***mapper_options*)
Bases: object

A Routes based dispatcher for CherryPy.

connect(*name*, *route*, *controller*, ***kwargs*)

find_handler(*path_info*)
Find the right page handler, and set request.config.

redirect(*url*)

cherrypy._cpdispatch.**VirtualHost**(*next_dispatcher=<cherrypy._cpdispatch.Dispatcher object>*, *use_x_forwarded_host=True*, ***domains*)
Select a different handler based on the Host header.

This can be useful when running multiple sites within one CP server. It allows several domains to point to different parts of a single website structure. For example:

```
http://www.domain.example  ->  root
http://www.domain2.example  ->  root/domain2/
http://www.domain2.example:443  ->  root/secure
```

can be accomplished via the following config:

```
[/]
request.dispatch = cherrypy.dispatch.VirtualHost(
    **{'www.domain2.example': '/domain2',
       'www.domain2.example:443': '/secure',
      })
```

next_dispatcher The next dispatcher object in the dispatch chain. The VirtualHost dispatcher adds a prefix to the URL and calls another dispatcher. Defaults to cherrypy.dispatch.Dispatcher().

use_x_forwarded_host If True (the default), any "X-Forwarded-Host" request header will be used instead of the "Host" header. This is commonly added by HTTP servers (such as Apache) when proxying.

****domains** A dict of {host header value: virtual prefix} pairs. The incoming "Host" request header is looked up in this dict, and, if a match is found, the corresponding "virtual prefix" value will be prepended to the URL path before calling the next dispatcher. Note that you often need separate entries for "example.com" and "www.example.com". In addition, "Host" headers may contain the port number.

cherrypy._cpdispatch.**XMLRPCDispatcher**(*next_dispatcher=<cherrypy._cpdispatch.Dispatcher object>*)

cherrypy._cpdispatch.**getargspec**(*callable*)

cherrypy._cpdispatch.**test_callable_spec**(*callable*, *callable_args*, *callable_kwargs*)
Inspect callable and test to see if the given args are suitable for it.

When an error occurs during the handler's invoking stage there are 2 erroneous cases: 1. Too many parameters passed to a function which doesn't define

one of *args or **kwargs.

2. Too little parameters are passed to the function.

There are 3 sources of parameters to a cherrypy handler. 1. query string parameters are passed as keyword parameters to the

handler.

2. body parameters are also passed as keyword parameters.
3. when partial matching occurs, the final path atoms are passed as positional args.

Both the query string and path atoms are part of the URI. If they are incorrect, then a 404 Not Found should be raised. Conversely the body parameters are part of the request; if they are invalid a 400 Bad Request.

cherrypy._cpdispatch.**validate_translator**(*t*)

cherrypy._cperror module

Exception classes for CherryPy.

CherryPy provides (and uses) exceptions for declaring that the HTTP response should be a status other than the default "200 OK". You can `raise` them like normal Python exceptions. You can also call them and they will raise themselves; this means you can set an *HTTPError* or *HTTPRedirect* as the *request.handler*.

Redirecting POST

When you GET a resource and are redirected by the server to another Location, there's generally no problem since GET is both a "safe method" (there should be no side-effects) and an "idempotent method" (multiple calls are no different than a single call).

POST, however, is neither safe nor idempotent–if you charge a credit card, you don't want to be charged twice by a redirect!

For this reason, *none* of the 3xx responses permit a user-agent (browser) to resubmit a POST on redirection without first confirming the action with the user:

300	Multiple Choices	Confirm with the user
301	Moved Permanently	Confirm with the user
302	Found (Object moved temporarily)	Confirm with the user
303	See Other	GET the new URI; no confirmation
304	Not modified	for conditional GET only; POST should not raise this error
305	Use Proxy	Confirm with the user
307	Temporary Redirect	Confirm with the user
308	Permanent Redirect	No confirmation

However, browsers have historically implemented these restrictions poorly; in particular, many browsers do not force the user to confirm 301, 302 or 307 when redirecting POST. For this reason, CherryPy defaults to 303, which most user-agents appear to have implemented correctly. Therefore, if you raise HTTPRedirect for a POST request, the user-agent will most likely attempt to GET the new URI (without asking for confirmation from the user). We realize this is confusing for developers, but it's the safest thing we could do. You are of course free to raise `HTTPRedirect(uri, status=302)` or any other 3xx status if you know what you're doing, but given the environment, we couldn't let any of those be the default.

Custom Error Handling

Anticipated HTTP responses

The 'error_page' config namespace can be used to provide custom HTML output for expected responses (like 404 Not Found). Supply a filename from which the output will be read. The contents will be interpolated with the values %(status)s, %(message)s, %(traceback)s, and %(version)s using plain old Python string formatting.

```
_cp_config = {
    'error_page.404': os.path.join(localDir, "static/index.html")
}
```

Beginning in version 3.1, you may also provide a function or other callable as an error_page entry. It will be passed the same status, message, traceback and version arguments that are interpolated into templates:

```
def error_page_402(status, message, traceback, version):
    return "Error %s - Well, I'm very sorry but you haven't paid!" % status
cherrypy.config.update({'error_page.402': error_page_402})
```

Also in 3.1, in addition to the numbered error codes, you may also supply "error_page.default" to handle all codes which do not have their own error_page entry.

Unanticipated errors

CherryPy also has a generic error handling mechanism: whenever an unanticipated error occurs in your code, it will call *Request.error_response* to set the response status, headers, and body. By default, this is the same output as *HTTPError(500)*. If you want to provide some other behavior, you generally replace "request.error_response".

Here is some sample code that shows how to display a custom error message and send an e-mail containing the error:

```
from cherrypy import _cperror

def handle_error():
    cherrypy.response.status = 500
    cherrypy.response.body = [
        "<html><body>Sorry, an error occurred</body></html>"
    ]
    sendMail('error@domain.com',
             'Error in your web app',
             _cperror.format_exc())

@cherrypy.config(**{'request.error_response': handle_error})
class Root:
    pass
```

Note that you have to explicitly set *response.body* and not simply return an error message as a result.

exception cherrypy._cperror.**CherryPyException**

Bases: Exception

A base class for CherryPy exceptions.

exception cherrypy._cperror.**HTTPError**(*status=500*, *message=None*)

Bases: *cherrypy._cperror.CherryPyException*

Exception used to return an HTTP error code (4xx-5xx) to the client.

This exception can be used to automatically send a response using a http status code, with an appropriate error page. It takes an optional `status` argument (which must be between 400 and 599); it defaults to 500 ("Internal Server Error"). It also takes an optional `message` argument, which will be returned in the response body. See RFC2616 for a complete list of available error codes and when to use them.

Examples:

```
raise cherrypy.HTTPError(403)
raise cherrypy.HTTPError(
    "403 Forbidden", "You are not allowed to access this resource.")
```

code = None
The integer HTTP status code.

get_error_page(**args*, ***kwargs*)

classmethod handle(*exception*, *status=500*, *message=''*)
Translate exception into an HTTPError.

reason = None
The HTTP Reason-Phrase string.

set_response()
Modify cherrypy.response status, headers, and body to represent self.

CherryPy uses this internally, but you can also use it to create an HTTPError object and set its output without *raising* the exception.

status = None
The HTTP status code. May be of type int or str (with a Reason-Phrase).

exception cherrypy._cperror.**HTTPRedirect**(*urls*, *status=None*, *encoding=None*)
Bases: *cherrypy._cperror.CherryPyException*

Exception raised when the request should be redirected.

This exception will force a HTTP redirect to the URL or URL's you give it. The new URL must be passed as the first argument to the Exception, e.g., HTTPRedirect(newUrl). Multiple URLs are allowed in a list. If a URL is absolute, it will be used as-is. If it is relative, it is assumed to be relative to the current cherrypy.request.path_info.

If one of the provided URL is a unicode object, it will be encoded using the default encoding or the one passed in parameter.

There are multiple types of redirect, from which you can select via the `status` argument. If you do not provide a `status` arg, it defaults to 303 (or 302 if responding with HTTP/1.0).

Examples:

```
raise cherrypy.HTTPRedirect("")
raise cherrypy.HTTPRedirect("/abs/path", 307)
raise cherrypy.HTTPRedirect(["path1", "path2?a=1&b=2"], 301)
```

See *Redirecting POST* for additional caveats.

default_status = 303

encoding = 'utf-8'
The encoding when passed urls are not native strings

set_response()
Modify cherrypy.response status, headers, and body to represent self.

CherryPy uses this internally, but you can also use it to create an HTTPRedirect object and set its output without *raising* the exception.

property status
The integer HTTP status code to emit.

urls = None
The list of URL's to emit.

exception cherrypy._cperror.**InternalRedirect**(*path*, *query_string=''*)
Bases: *cherrypy._cperror.CherryPyException*

Exception raised to switch to the handler for a different URL.

This exception will redirect processing to another path within the site (without informing the client). Provide the new path as an argument when raising the exception. Provide any params in the querystring for the new URL.

exception cherrypy._cperror.**NotFound**(*path=None*)
Bases: *cherrypy._cperror.HTTPError*

Exception raised when a URL could not be mapped to any handler (404).

This is equivalent to raising *HTTPError("404 Not Found")*.

cherrypy._cperror.**_be_ie_unfriendly**(*status*)

cherrypy._cperror.**bare_error**(*extrabody=None*)
Produce status, headers, body for a critical error.

Returns a triple without calling any other questionable functions, so it should be as error-free as possible. Call it from an HTTP server if you get errors outside of the request.

If extrabody is None, a friendly but rather unhelpful error message is set in the body. If extrabody is a string, it will be appended as-is to the body.

cherrypy._cperror.**clean_headers**(*status*)
Remove any headers which should not apply to an error response.

cherrypy._cperror.**format_exc**(*exc=None*)
Return exc (or sys.exc_info if None), formatted.

cherrypy._cperror.**get_error_page**(*status*, ***kwargs*)
Return an HTML page, containing a pretty error response.

status should be an int or a str. kwargs will be interpolated into the page template.

cherrypy._cplogging module

Simple config

Although CherryPy uses the Python logging module, it does so behind the scenes so that simple logging is simple, but complicated logging is still possible. "Simple" logging means that you can log to the screen (i.e. console/stdout) or to a file, and that you can easily have separate error and access log files.

Here are the simplified logging settings. You use these by adding lines to your config file or dict. You should set these at either the global level or per application (see next), but generally not both.

- log.screen: Set this to True to have both "error" and "access" messages printed to stdout.
- log.access_file: Set this to an absolute filename where you want "access" messages written.

- `log.error_file`: Set this to an absolute filename where you want "error" messages written.

Many events are automatically logged; to log your own application events, call `cherrypy.log()`.

Architecture

Separate scopes

CherryPy provides log managers at both the global and application layers. This means you can have one set of logging rules for your entire site, and another set of rules specific to each application. The global log manager is found at `cherrypy.log()`, and the log manager for each application is found at *`app.log`*. If you're inside a request, the latter is reachable from `cherrypy.request.app.log`; if you're outside a request, you'll have to obtain a reference to the `app`: either the return value of *`tree.mount()`* or, if you used *`quickstart()`* instead, via `cherrypy.tree.apps['/']`.

By default, the global logs are named "cherrypy.error" and "cherrypy.access", and the application logs are named "cherrypy.error.2378745" and "cherrypy.access.2378745" (the number is the id of the Application object). This means that the application logs "bubble up" to the site logs, so if your application has no log handlers, the site-level handlers will still log the messages.

Errors vs. Access

Each log manager handles both "access" messages (one per HTTP request) and "error" messages (everything else). Note that the "error" log is not just for errors! The format of access messages is highly formalized, but the error log isn't–it receives messages from a variety of sources (including full error tracebacks, if enabled).

If you are logging the access log and error log to the same source, then there is a possibility that a specially crafted error message may replicate an access log message as described in CWE-117. In this case it is the application developer's responsibility to manually escape data before using CherryPy's log() functionality, or they may create an application that is vulnerable to CWE-117. This would be achieved by using a custom handler escape any special characters, and attached as described below.

Custom Handlers

The simple settings above work by manipulating Python's standard `logging` module. So when you need something more complex, the full power of the standard module is yours to exploit. You can borrow or create custom handlers, formats, filters, and much more. Here's an example that skips the standard FileHandler and uses a RotatingFileHandler instead:

```
#python
log = app.log

# Remove the default FileHandlers if present.
log.error_file = ""
log.access_file = ""

maxBytes = getattr(log, "rot_maxBytes", 10000000)
backupCount = getattr(log, "rot_backupCount", 1000)

# Make a new RotatingFileHandler for the error log.
fname = getattr(log, "rot_error_file", "error.log")
h = handlers.RotatingFileHandler(fname, 'a', maxBytes, backupCount)
h.setLevel(DEBUG)
```

(continues on next page)

(continued from previous page)

```
h.setFormatter(_cplogging.logfmt)
log.error_log.addHandler(h)

# Make a new RotatingFileHandler for the access log.
fname = getattr(log, "rot_access_file", "access.log")
h = handlers.RotatingFileHandler(fname, 'a', maxBytes, backupCount)
h.setLevel(DEBUG)
h.setFormatter(_cplogging.logfmt)
log.access_log.addHandler(h)
```

The `rot_*` attributes are pulled straight from the application log object. Since "log.*" config entries simply set attributes on the log object, you can add custom attributes to your heart's content. Note that these handlers are used ''instead" of the default, simple handlers outlined above (so don't set the "log.error_file" config entry, for example).

class cherrypy._cplogging.**LazyRfc3339UtcTime**
Bases: object

class cherrypy._cplogging.**LogManager**(*appid=None*, *logger_root='cherrypy'*)
Bases: object

An object to assist both simple and advanced logging.

`cherrypy.log` is an instance of this class.

_add_builtin_file_handler(*log*, *fname*)

_get_builtin_handler(*log*, *key*)

_set_file_handler(*log*, *filename*)

_set_screen_handler(*log*, *enable*, *stream=None*)

_set_wsgi_handler(*log*, *enable*)

access()
Write to the access log (in Apache/NCSA Combined Log format).

See the apache documentation for format details.

CherryPy calls this automatically for you. Note there are no arguments; it collects the data itself from *cherrypy.request*.

Like Apache started doing in 2.0.46, non-printable and other special characters in %r (and we expand that to all parts) are escaped using xhh sequences, where hh stands for the hexadecimal representation of the raw byte. Exceptions from this rule are " and , which are escaped by prepending a backslash, and all whitespace characters, which are written in their C-style notation (n, t, etc).

property access_file
The filename for self.access_log.

If you set this to a string, it'll add the appropriate FileHandler for you. If you set it to `None` or `''`, it will remove the handler.

access_log = None
The actual `logging.Logger` instance for access messages.

access_log_format = '{h} {l} {u} {t} "{r}" {s} {b} "{f}" "{a}"'

appid = None
The id() of the Application object which owns this log manager. If this is a global log manager, appid is None.

error(*msg=''*, *context=''*, *severity=20*, *traceback=False*)
Write the given `msg` to the error log.

This is not just for errors! Applications may call this at any time to log application-specific information.

If `traceback` is True, the traceback of the current exception (if any) will be appended to `msg`.

property error_file
The filename for self.error_log.

If you set this to a string, it'll add the appropriate FileHandler for you. If you set it to `None` or `''`, it will remove the handler.

error_log = None
The actual `logging.Logger` instance for error messages.

logger_root = None
The "top-level" logger name.

This string will be used as the first segment in the Logger names. The default is "cherrypy", for example, in which case the Logger names will be of the form:

```
cherrypy.error.<appid>
cherrypy.access.<appid>
```

reopen_files()
Close and reopen all file handlers.

property screen
Turn stderr/stdout logging on or off.

If you set this to True, it'll add the appropriate StreamHandler for you. If you set it to False, it will remove the handler.

time()
Return now() in Apache Common Log Format (no timezone).

property wsgi
Write errors to wsgi.errors.

If you set this to True, it'll add the appropriate *WSGIErrorHandler* for you (which writes errors to `wsgi.errors`). If you set it to False, it will remove the handler.

class cherrypy._cplogging.**NullHandler**(*level=0*)
Bases: `logging.Handler`

A no-op logging handler to silence the logging.lastResort handler.

createLock()
Acquire a thread lock for serializing access to the underlying I/O.

emit(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

handle(*record*)
Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock. Returns whether the filter passed the record for emission.

class cherrypy._cplogging.**WSGIErrorHandler**(*level=0*)
Bases: logging.Handler

A handler class which writes logging records to environ['wsgi.errors'].

emit(*record*)
Emit a record.

flush()
Flushes the stream.

cherrypy._cpmodpy module

Native adapter for serving CherryPy via mod_python

Basic usage:

Application in a module called myapp.py

import cherrypy

class Root: @cherrypy.expose def index(self):

return 'Hi there, Ho there, Hey there'

We will use this method from the mod_python configuration # as the entry point to our application def setup_server():

cherrypy.tree.mount(Root()) cherrypy.config.update({'environment': 'production',

'log.screen': False, 'show_tracebacks': False})

or a file that will be loaded at # apache startup

Start DocumentRoot "/" Listen 8080 LoadModule python_module /usr/lib/apache2/modules/mod_python.so

<Location "/"> PythonPath "sys.path+['/path/to/my/application']" SetHandler python-program PythonHandler cherrypy._cpmodpy::handler PythonOption cherrypy.setup myapp::setup_server PythonDebug On

</Location> # End

The actual path to your mod_python.so is dependent on your environment. In this case we suppose a global mod_python installation on a Linux distribution such as Ubuntu.

We do set the PythonPath configuration setting so that your application can be found by from the user running the apache2 instance. Of course if your application resides in the global site-package this won't be needed.

Then restart apache2 and access http://127.0.0.1:8080

class cherrypy._cpmodpy.**ModPythonServer**(*loc='/'*, *port=80*, *opts=None*, *apache_path='apache'*, *handler='cherrypy._cpmodpy::handler'*)
Bases: object

start()

stop()

template = '\n# Apache2 server configuration file for running CherryPy with mod_python

class cherrypy._cpmodpy.**_ReadOnlyRequest**(*req*)
Bases: object

expose = ('read', 'readline', 'readlines')

cherrypy._cpmodpy.**handler**(*req*)

cherrypy._cpmodpy.**popen**(*fullcmd*)

cherrypy._cpmodpy.**read_process**(*cmd*, *args=''*)

cherrypy._cpmodpy.**send_response**(*req*, *status*, *headers*, *body*, *stream=False*)

cherrypy._cpmodpy.**setup**(*req*)

cherrypy._cpnative_server module

Native adapter for serving CherryPy via its builtin server.

class cherrypy._cpnative_server.**CPHTTPServer**(*server_adapter=<cherrypy._cpserver.Server object>*)

Bases: cheroot.server.HTTPServer

Wrapper for cheroot.server.HTTPServer.

cheroot has been designed to not reference CherryPy in any way, so that it can be used in other frameworks and applications. Therefore, we wrap it here, so we can apply some attributes from config -> cherrypy.server -> HTTPServer.

class cherrypy._cpnative_server.**NativeGateway**(*req*)

Bases: cheroot.server.Gateway

Native gateway implementation allowing to bypass WSGI.

recursive = False

respond()

Obtain response from CherryPy machinery and then send it.

send_response(*status*, *headers*, *body*)

Send response to HTTP request.

cherrypy._cpreqbody module

Request body processing for CherryPy.

New in version 3.2.

Application authors have complete control over the parsing of HTTP request entities. In short, *cherrypy.request.body* is now always set to an instance of *RequestBody*, and *that* class is a subclass of *Entity*.

When an HTTP request includes an entity body, it is often desirable to provide that information to applications in a form other than the raw bytes. Different content types demand different approaches. Examples:

- For a GIF file, we want the raw bytes in a stream.
- An HTML form is better parsed into its component fields, and each text field decoded from bytes to unicode.
- A JSON body should be deserialized into a Python dict or list.

When the request contains a Content-Type header, the media type is used as a key to look up a value in the *request.body.processors* dict. If the full media type is not found, then the major type is tried; for example, if no processor is found for the 'image/jpeg' type, then we look for a processor for the 'image' types altogether. If neither the full type nor the major type has a matching processor, then a default processor is used (*default_proc*). For most types, this means no processing is done, and the body is left unread as a raw byte stream. Processors are configurable in an 'on_start_resource' hook.

Some processors, especially those for the 'text' types, attempt to decode bytes to unicode. If the Content-Type request header includes a 'charset' parameter, this is used to decode the entity. Otherwise, one or more default charsets may be attempted, although this decision is up to each processor. If a processor successfully decodes an Entity or Part, it should set the *charset* attribute on the Entity or Part to the name of the successful charset, so that applications can easily re-encode or transcode the value if they wish.

If the Content-Type of the request entity is of major type 'multipart', then the above parsing process, and possibly a decoding process, is performed for each part.

For both the full entity and multipart parts, a Content-Disposition header may be used to fill *name* and *filename* attributes on the request.body or the Part.

Custom Processors

You can add your own processors for any specific or major MIME type. Simply add it to the `processors` dict in a hook/tool that runs at `on_start_resource` or `before_request_body`. Here's the built-in JSON tool for an example:

```
def json_in(force=True, debug=False):
    request = cherrypy.serving.request
    def json_processor(entity):
        '''Read application/json data into request.json.'''
        if not entity.headers.get("Content-Length", ""):
            raise cherrypy.HTTPError(411)

        body = entity.fp.read()
        try:
            request.json = json_decode(body)
        except ValueError:
            raise cherrypy.HTTPError(400, 'Invalid JSON document')
    if force:
        request.body.processors.clear()
        request.body.default_proc = cherrypy.HTTPError(
            415, 'Expected an application/json content type')
    request.body.processors['application/json'] = json_processor
```

We begin by defining a new `json_processor` function to stick in the `processors` dictionary. All processor functions take a single argument, the `Entity` instance they are to process. It will be called whenever a request is received (for those URI's where the tool is turned on) which has a `Content-Type` of "application/json".

First, it checks for a valid `Content-Length` (raising 411 if not valid), then reads the remaining bytes on the socket. The `fp` object knows its own length, so it won't hang waiting for data that never arrives. It will return when all data has been read. Then, we decode those bytes using Python's built-in `json` module, and stick the decoded result onto `request.json` . If it cannot be decoded, we raise 400.

If the "force" argument is True (the default), the `Tool` clears the `processors` dict so that request entities of other `Content-Types` aren't parsed at all. Since there's no entry for those invalid MIME types, the `default_proc` method of `cherrypy.request.body` is called. But this does nothing by default (usually to provide the page handler an opportunity to handle it.) But in our case, we want to raise 415, so we replace `request.body.default_proc` with the error (`HTTPError` instances, when called, raise themselves).

If we were defining a custom processor, we can do so without making a `Tool`. Just add the config entry:

```
request.body.processors = {'application/json': json_processor}
```

Note that you can only replace the `processors` dict wholesale this way, not update the existing one.

class cherrypy._cpreqbody.**Entity**(*fp*, *headers*, *params=None*, *parts=None*)
Bases: object

An HTTP request body, or MIME multipart body.

This class collects information about the HTTP request entity. When a given entity is of MIME type "multipart", each part is parsed into its own Entity instance, and the set of parts stored in *entity.parts*.

Between the before_request_body and before_handler tools, CherryPy tries to process the request body (if any) by calling *request.body.process*. This uses the content_type of the Entity to look up a suitable processor in *Entity.processors*, a dict. If a matching processor cannot be found for the complete Content-Type, it tries again using the major type. For example, if a request with an entity of type "image/jpeg" arrives, but no processor can be found for that complete type, then one is sought for the major type "image". If a processor is still not found, then the *default_proc* method of the Entity is called (which does nothing by default; you can override this too).

CherryPy includes processors for the "application/x-www-form-urlencoded" type, the "multipart/form-data" type, and the "multipart" major type. CherryPy 3.2 processes these types almost exactly as older versions. Parts are passed as arguments to the page handler using their Content-Disposition.name if given, otherwise in a generic "parts" argument. Each such part is either a string, or the *Part* itself if it's a file. (In this case it will have file and filename attributes, or possibly a value attribute). Each Part is itself a subclass of Entity, and has its own process method and processors dict.

There is a separate processor for the "multipart" major type which is more flexible, and simply stores all multipart parts in *request.body.parts*. You can enable it with:

```
cherrypy.request.body.processors['multipart'] = _cpreqbody.process_
↪multipart
```

in an on_start_resource tool.

attempt_charsets = ['utf-8']
A list of strings, each of which should be a known encoding.

When the Content-Type of the request body warrants it, each of the given encodings will be tried in order. The first one to successfully decode the entity without raising an error is stored as *entity.charset*. This defaults to ['utf-8'] (plus 'ISO-8859-1' for "text/*" types, as required by HTTP/1.1), but ['us-ascii', 'utf-8'] for multipart parts.

charset = None
The successful decoding; see "attempt_charsets" above.

content_type = None
The value of the Content-Type request header.

If the Entity is part of a multipart payload, this will be the Content-Type given in the MIME headers for this part.

decode_entity(*value*)
Return a given byte encoded value as a string

default_content_type = 'application/x-www-form-urlencoded'
This defines a default Content-Type to use if no Content-Type header is given. The empty string is used for RequestBody, which results in the request body not being read or parsed at all. This is by design; a missing Content-Type header in the HTTP request entity is an error at best, and a security hole at worst. For multipart parts, however, the MIME spec declares that a part with no Content-Type defaults to "text/plain" (see *Part*).

default_proc()
Called if a more-specific processor is not found for the Content-Type.

filename = None
The `Content-Disposition.filename` header, if available.

fp = None
The readable socket file object.

fullvalue()
Return this entity as a string, whether stored in a file or not.

headers = None
A dict of request/multipart header names and values.

This is a copy of the `request.headers` for the `request.body`; for multipart parts, it is the set of headers for that part.

length = None
The value of the `Content-Length` header, if provided.

make_file()
Return a file-like object into which the request body will be read.

By default, this will return a TemporaryFile. Override as needed. See also *cherrypy._cpreqbody.Part.maxrambytes*.

name = None
The "name" parameter of the `Content-Disposition` header, if any.

next()

params = None
If the request Content-Type is 'application/x-www-form-urlencoded' or multipart, this will be a dict of the params pulled from the entity body; that is, it will be the portion of request.params that come from the message body (sometimes called "POST params", although they can be sent with various HTTP method verbs). This value is set between the 'before_request_body' and 'before_handler' hooks (assuming that process_request_body is True).

part_class
The class used for multipart parts.

You can replace this with custom subclasses to alter the processing of multipart parts.

alias of *cherrypy._cpreqbody.Part*

parts = None
A list of Part instances if `Content-Type` is of major type "multipart".

process()
Execute the best-match processor for the given media type.

processors = {'application/x-www-form-urlencoded': <function process_urlencoded>, 'mu
A dict of Content-Type names to processor methods.

read(*size=None*, *fp_out=None*)

read_into_file(*fp_out=None*)
Read the request body into fp_out (or make_file() if None).

Return fp_out.

readline(*size=None*)

readlines(*sizehint=None*)

class cherrypy._cpreqbody.**Part**(*fp*, *headers*, *boundary*)
Bases: *cherrypy._cpreqbody.Entity*

A MIME part entity, part of a multipart entity.

attempt_charsets = ['us-ascii', 'utf-8']
A list of strings, each of which should be a known encoding.

When the Content-Type of the request body warrants it, each of the given encodings will be tried in order. The first one to successfully decode the entity without raising an error is stored as *entity.charset*. This defaults to `['utf-8']` (plus 'ISO-8859-1' for "text/*" types, as required by HTTP/1.1), but `['us-ascii', 'utf-8']` for multipart parts.

boundary = None
The MIME multipart boundary.

default_content_type = 'text/plain'
This defines a default `Content-Type` to use if no Content-Type header is given. The empty string is used for RequestBody, which results in the request body not being read or parsed at all. This is by design; a missing `Content-Type` header in the HTTP request entity is an error at best, and a security hole at worst. For multipart parts, however (this class), the MIME spec declares that a part with no Content-Type defaults to "text/plain".

default_proc()
Called if a more-specific processor is not found for the `Content-Type`.

classmethod from_fp(*fp*, *boundary*)

maxrambytes = 1000
The threshold of bytes after which point the `Part` will store its data in a file (generated by `make_file`) instead of a string. Defaults to 1000, just like the `cgi` module in Python's standard library.

classmethod read_headers(*fp*)

read_into_file(*fp_out=None*)
Read the request body into fp_out (or make_file() if None).

Return fp_out.

read_lines_to_boundary(*fp_out=None*)
Read bytes from self.fp and return or write them to a file.

If the 'fp_out' argument is None (the default), all bytes read are returned in a single byte string.

If the 'fp_out' argument is not None, it must be a file-like object that supports the 'write' method; all bytes read will be written to the fp, and that fp is returned.

class cherrypy._cpreqbody.**RequestBody**(*fp*, *headers*, *params=None*, *request_params=None*)
Bases: *cherrypy._cpreqbody.Entity*

The entity of the HTTP request.

bufsize = 8192
The buffer size used when reading the socket.

default_content_type = ''
This defines a default `Content-Type` to use if no Content-Type header is given. The empty string is used for RequestBody, which results in the request body not being read or parsed at all. This is by design; a missing `Content-Type` header in the HTTP request entity is an error at best, and a security hole at worst. For multipart parts, however, the MIME spec declares that a part with no Content-Type defaults to "text/plain" (see *Part*).

maxbytes = None
Raise `MaxSizeExceeded` if more bytes than this are read from the socket.

process()
Process the request entity based on its Content-Type.

class cherrypy._cpreqbody.**SizedReader**(*fp*, *length*, *maxbytes*, *bufsize=8192*, *has_trailers=False*)
Bases: `object`

finish()

read(*size=None*, *fp_out=None*)
Read bytes from the request body and return or write them to a file.

A number of bytes less than or equal to the 'size' argument are read off the socket. The actual number of bytes read are tracked in self.bytes_read. The number may be smaller than 'size' when 1) the client sends fewer bytes, 2) the 'Content-Length' request header specifies fewer bytes than requested, or 3) the number of bytes read exceeds self.maxbytes (in which case, 413 is raised).

If the 'fp_out' argument is None (the default), all bytes read are returned in a single byte string.

If the 'fp_out' argument is not None, it must be a file-like object that supports the 'write' method; all bytes read will be written to the fp, and None is returned.

readline(*size=None*)
Read a line from the request body and return it.

readlines(*sizehint=None*)
Read lines from the request body and return them.

cherrypy._cpreqbody.**_old_process_multipart**(*entity*)
The behavior of 3.2 and lower. Deprecated and will be changed in 3.3.

cherrypy._cpreqbody.**process_multipart**(*entity*)
Read all multipart parts into entity.parts.

cherrypy._cpreqbody.**process_multipart_form_data**(*entity*)
Read all multipart/form-data parts into entity.parts or entity.params.

cherrypy._cpreqbody.**process_urlencoded**(*entity*)
Read application/x-www-form-urlencoded data into entity.params.

cherrypy._cpreqbody.**unquote_plus**(*bs*)
Bytes version of urllib.parse.unquote_plus.

cherrypy._cprequest module

class cherrypy._cprequest.**Hook**(*callback*, *failsafe=None*, *priority=None*, ***kwargs*)
Bases: `object`

A callback and its metadata: failsafe, priority, and kwargs.

callback = None
The bare callable that this Hook object is wrapping, which will be called when the Hook is called.

failsafe = False
If True, the callback is guaranteed to run even if other callbacks from the same call point raise exceptions.

kwargs = {}
A set of keyword arguments that will be passed to the callable on each call.

priority = 50
Defines the order of execution for a list of Hooks. Priority numbers should be limited to the closed interval [0, 100], but values outside this range are acceptable, as are fractional values.

class cherrypy._cprequest.**HookMap**(*points=None*)
Bases: dict

A map of call points to lists of callbacks (Hook objects).

attach(*point*, *callback*, *failsafe=None*, *priority=None*, ***kwargs*)
Append a new Hook made from the supplied arguments.

copy() → a shallow copy of D

run(*point*)
Execute all registered Hooks (callbacks) for the given point.

classmethod run_hooks(*hooks*)
Execute the indicated hooks, trapping errors.

Hooks with `.failsafe == True` are guaranteed to run even if others at the same hookpoint fail. In this case, log the failure and proceed on to the next hook. The only way to stop all processing from one of these hooks is to raise a BaseException like SystemExit or KeyboardInterrupt and stop the whole server.

class cherrypy._cprequest.**LazyUUID4**
Bases: object

property uuid4
Provide unique id on per-request basis using UUID4.

It's evaluated lazily on render.

class cherrypy._cprequest.**Request**(*local_host*, *remote_host*, *scheme='http'*, *server_protocol='HTTP/1.1'*)
Bases: object

An HTTP request.

This object represents the metadata of an HTTP request message; that is, it contains attributes which describe the environment in which the request URL, headers, and body were sent (if you want tools to interpret the headers and body, those are elsewhere, mostly in Tools). This 'metadata' consists of socket data, transport characteristics, and the Request-Line. This object also contains data regarding the configuration in effect for the given URL, and the execution plan for generating a response.

_do_respond(*path_info*)

app = None
The cherrypy.Application object which is handling this request.

base = ''
//host) portion of the requested URL. In some cases (e.g. when proxying via mod_rewrite), this may contain path segments which cherrypy.url uses when constructing url's, but which otherwise are ignored by CherryPy. Regardless, this value MUST NOT end in a slash.

Type The (scheme

body = None
If the request Content-Type is 'application/x-www-form-urlencoded' or multipart, this will be None. Otherwise, this will be an instance of *RequestBody* (which you can .read()); this value is set between the 'before_request_body' and 'before_handler' hooks (assuming that process_request_body is True).

close()
Run cleanup code. (Core)

closed = False
True once the close method has been called, False otherwise.

config = None
A flat dict of all configuration entries which apply to the current request. These entries are collected from global config, application config (based on request.path_info), and from handler config (exactly how is governed by the request.dispatch object in effect for this request; by default, handler config can be attached anywhere in the tree between request.app.root and the final handler, and inherits downward).

cookie = {}
See help(Cookie).

dispatch = <cherrypy._cpdispatch.Dispatcher object>
The object which looks up the 'page handler' callable and collects config for the current request based on the path_info, other request attributes, and the application architecture. The core calls the dispatcher as early as possible, passing it a 'path_info' argument.

The default dispatcher discovers the page handler by matching path_info to a hierarchical arrangement of objects, starting at request.app.root. See help(cherrypy.dispatch) for more information.

error_page = {}
response filename or callable} pairs.

The error code must be an int representing a given HTTP error code, or the string 'default', which will be used if no matching entry is found for a given numeric code.

If a filename is provided, the file should contain a Python string- formatting template, and can expect by default to receive format values with the mapping keys %(status)s, %(message)s, %(traceback)s, and %(version)s. The set of format mappings can be extended by overriding HTTPError.set_response.

If a callable is provided, it will be called by default with keyword arguments 'status', 'message', 'traceback', and 'version', as for a string-formatting template. The callable must return a string or iterable of strings which will be set to response.body. It may also override headers or perform any other processing.

If no entry is given for an error code, and no 'default' entry exists, a default template will be used.

Type A dict of {error code

error_response()
The no-arg callable which will handle unexpected, untrapped errors during request processing. This is not used for expected exceptions (like NotFound, HTTPError, or HTTPRedirect) which are raised in response to expected conditions (those should be customized either via request.error_page or by overriding HTTPError.set_response). By default, error_response uses HTTPError(500) to return a generic error response to the user-agent.

get_resource(*path*)
Call a dispatcher (which sets self.handler and .config). (Core)

handle_error()
Handle the last unanticipated exception. (Core)

handler = None
The function, method, or other callable which CherryPy will call to produce the response. The discovery of the handler and the arguments it will receive are determined by the request.dispatch object. By default, the handler is discovered by walking a tree of objects starting at request.app.root, and is then passed all HTTP params (from the query string and POST body) as keyword arguments.

header_list = []
A list of the HTTP request headers as (name, value) tuples. In general, you should use request.headers (a dict) instead.

headers = {}
A dict-like object containing the request headers. Keys are header names (in Title-Case format); however, you may get and set them in a case-insensitive manner. That is, headers['Content-Type'] and headers['content-type'] refer to the same value. Values are header values (decoded according to **RFC 2047** if necessary). See also: httputil.HeaderMap, httputil.HeaderElement.

hooks = {'after_error_response': [], 'before_error_response': [], 'before_finalize':
[hook, ...]}. Each key is a str naming the hook point, and each value is a list of hooks which will be called at that hook point during this request. The list of hooks is generally populated as early as possible (mostly from Tools specified in config), but may be extended at any time. See also: _cprequest.Hook, _cprequest.HookMap, and cherrypy.tools.

Type A HookMap (dict-like object) of the form

Type {hookpoint

is_index = None
This will be True if the current request is mapped to an 'index' resource handler (also, a 'default' handler if path_info ends with a slash). The value may be used to automatically redirect the user-agent to a 'more canonical' URL which either adds or removes the trailing slash. See cherrypy.tools.trailing_slash.

local = httputil.Host('127.0.0.1', 80, '127.0.0.1')
An httputil.Host(ip, port, hostname) object for the server socket.

login = None
When authentication is used during the request processing this is set to 'False' if it failed and to the 'username' value if it succeeded. The default 'None' implies that no authentication happened.

method = 'GET'
Indicates the HTTP method to be performed on the resource identified by the Request-URI. Common methods include GET, HEAD, POST, PUT, and DELETE. CherryPy allows any extension method; however, various HTTP servers and gateways may restrict the set of allowable methods. CherryPy applications SHOULD restrict the set (on a per-URI basis).

methods_with_bodies = ('POST', 'PUT', 'PATCH')
A sequence of HTTP methods for which CherryPy will automatically attempt to read a body from the rfile. If you are going to change this property, modify it on the configuration (recommended) or on the "hook point" `on_start_resource`.

namespaces = {'error_page': <function error_page_namespace>, 'hooks': <function hook

params = {}
A dict which combines query string (GET) and request entity (POST) variables. This is populated in two stages: GET params are added before the 'on_start_resource' hook, and POST params are added between the 'before_request_body' and 'before_handler' hooks.

path_info = '/'
The 'relative path' portion of the Request-URI. This is relative to the script_name ('mount point') of the application which is handling this request.

prev = None
The previous Request object (if any). This should be None unless we are processing an InternalRedirect.

process_headers()
Parse HTTP header data into Python structures. (Core)

process_query_string()
Parse the query string into Python structures. (Core)

process_request_body = True
If True, the rfile (if any) is automatically read and parsed, and the result placed into request.params or

request.body.

protocol = (1, 1)
The HTTP protocol version corresponding to the set of features which should be allowed in the response. If BOTH the client's request message AND the server's level of HTTP compliance is HTTP/1.1, this attribute will be the tuple (1, 1). If either is 1.0, this attribute will be the tuple (1, 0). Lower HTTP protocol versions are not explicitly supported.

query_string = ''
The query component of the Request-URI, a string of information to be interpreted by the resource. The query portion of a URI follows the path component, and is separated by a '?'. For example, the URI 'http://www.cherrypy.org/wiki?a=3&b=4' has the query component, 'a=3&b=4'.

query_string_encoding = 'utf8'
The encoding expected for query string arguments after % HEX HEX decoding). If a query string is provided that cannot be decoded with this encoding, 404 is raised (since technically it's a different URI). If you want arbitrary encodings to not error, set this to 'Latin-1'; you can then encode back to bytes and re-decode to whatever encoding you like later.

remote = httputil.Host('127.0.0.1', 1111, '127.0.0.1')
An httputil.Host(ip, port, hostname) object for the client socket.

request_line = ''
The complete Request-Line received from the client. This is a single string consisting of the request method, URI, and protocol version (joined by spaces). Any final CRLF is removed.

respond(*path_info*)
Generate a response for the resource at self.path_info. (Core)

rfile = None
If the request included an entity (body), it will be available as a stream in this attribute. However, the rfile will normally be read for you between the 'before_request_body' hook and the 'before_handler' hook, and the resulting string is placed into either request.params or the request.body attribute.

You may disable the automatic consumption of the rfile by setting request.process_request_body to False, either in config for the desired path, or in an 'on_start_resource' or 'before_request_body' hook.

WARNING: In almost every case, you should not attempt to read from the rfile stream after CherryPy's automatic mechanism has read it. If you turn off the automatic parsing of rfile, you should read exactly the number of bytes specified in request.headers['Content-Length']. Ignoring either of these warnings may result in a hung request thread or in corruption of the next (pipelined) request.

run(*method*, *path*, *query_string*, *req_protocol*, *headers*, *rfile*)
Process the Request. (Core)

method, path, query_string, and req_protocol should be pulled directly from the Request-Line (e.g. "GET /path?key=val HTTP/1.0").

path This should be %XX-unquoted, but query_string should not be.

When using Python 2, they both MUST be byte strings, not unicode strings.

When using Python 3, they both MUST be unicode strings, not byte strings, and preferably not bytes x00-xFF disguised as unicode.

headers A list of (name, value) tuples.

rfile A file-like object containing the HTTP request entity.

When run() is done, the returned object should have 3 attributes:

- status, e.g. "200 OK"

- header_list, a list of (name, value) tuples
- body, an iterable yielding strings

Consumer code (HTTP servers) should then access these response attributes to build the outbound stream.

scheme = 'http'
The protocol used between client and server. In most cases, this will be either 'http' or 'https'.

script_name = ''
The 'mount point' of the application which is handling this request.

This attribute MUST NOT end in a slash. If the script_name refers to the root of the URI, it MUST be an empty string (not "/").

server_protocol = 'HTTP/1.1'
The HTTP version for which the HTTP server is at least conditionally compliant.

show_mismatched_params = True
If True, mismatched parameters encountered during PageHandler invocation processing will be included in the response body.

show_tracebacks = True
If True, unexpected errors encountered during request processing will include a traceback in the response body.

stage = None
A string containing the stage reached in the request-handling process. This is useful when debugging a live server with hung requests.

throw_errors = False
If True, Request.run will not trap any errors (except HTTPRedirect and HTTPError, which are more properly called 'exceptions', not errors).

throws = (<class 'KeyboardInterrupt'>, <class 'SystemExit'>, <class 'cherrypy._cperror
The sequence of exceptions which Request.run does not trap.

toolmaps = {}
A nested dict of all Toolboxes and Tools in effect for this request, of the form: {Toolbox.namespace: {Tool.name: config dict}}.

unique_id = None
A lazy object generating and memorizing UUID4 on `str()` render.

class cherrypy._cprequest.**Response**
Bases: object

An HTTP Response, including status, headers, and body.

_flush_body()
Discard self.body but consume any generator such that any finalization can occur, such as is required by caching.tee_output().

body
The body (entity) of the HTTP response.

collapse_body()
Collapse self.body to a single string; replace it and return it.

cookie = {}
See help(Cookie).

finalize()
Transform headers (and cookies) into self.header_list. (Core)

header_list = []
A list of the HTTP response headers as (name, value) tuples. In general, you should use response.headers (a dict) instead. This attribute is generated from response.headers and is not valid until after the finalize phase.

headers = {}
A dict-like object containing the response headers. Keys are header names (in Title-Case format); however, you may get and set them in a case-insensitive manner. That is, headers['Content-Type'] and headers['content-type'] refer to the same value. Values are header values (decoded according to **RFC 2047** if necessary).

See also:

classes `HeaderMap`, `HeaderElement`

status = ''
The HTTP Status-Code and Reason-Phrase.

stream = False
If False, buffer the response body.

time = None
The value of time.time() when created. Use in HTTP dates.

class cherrypy._cprequest.**ResponseBody**
Bases: `object`

The body of the HTTP response (the response entity).

unicode_err = 'Page handlers MUST return bytes. Use tools.encode if you wish to return

cherrypy._cprequest.**error_page_namespace**(*k*, *v*)
Attach error pages declared in config.

cherrypy._cprequest.**hooks_namespace**(*k*, *v*)
Attach bare hooks declared in config.

cherrypy._cprequest.**request_namespace**(*k*, *v*)
Attach request attributes declared in config.

cherrypy._cprequest.**response_namespace**(*k*, *v*)
Attach response attributes declared in config.

cherrypy._cpserver module

Manage HTTP servers with CherryPy.

class cherrypy._cpserver.**Server**
Bases: *cherrypy.process.servers.ServerAdapter*

An adapter for an HTTP server.

You can set attributes (like socket_host and socket_port) on *this* object (which is probably cherrypy.server), and call quickstart. For example:

```
cherrypy.server.socket_port = 80
cherrypy.quickstart()
```

_socket_host = '127.0.0.1'

accepted_queue_size = -1
The maximum number of requests which will be queued up before the server refuses to accept it (default -1, meaning no limit).

accepted_queue_timeout = 10
The timeout in seconds for attempting to add a request to the queue when the queue is full (default 10).

base()
Return the base for this server.

e.i. scheme://host[:port] or sock file

property bind_addr
Return bind address.

A (host, port) tuple for TCP sockets or a str for Unix domain sockts.

httpserver_from_self(*httpserver=None*)
Return a (httpserver, bind_addr) pair based on self attributes.

instance = None
If not None, this should be an HTTP server instance (such as cheroot.wsgi.Server) which cherrypy.server will control. Use this when you need more control over object instantiation than is available in the various configuration options.

max_request_body_size = 104857600
The maximum number of bytes allowable in the request body. If exceeded, the HTTP server should return "413 Request Entity Too Large".

max_request_header_size = 512000
The maximum number of bytes allowable in the request headers. If exceeded, the HTTP server should return "413 Request Entity Too Large".

nodelay = True
If True (the default since 3.1), sets the TCP_NODELAY socket option.

peercreds = False
If True, peer cred lookup for UNIX domain socket will put to WSGI env.

This information will then be available through WSGI env vars: * X_REMOTE_PID * X_REMOTE_UID * X_REMOTE_GID

peercreds_resolve = False
If True, username/group will be looked up in the OS from peercreds.

This information will then be available through WSGI env vars: * REMOTE_USER * X_REMOTE_USER * X_REMOTE_GROUP

protocol_version = 'HTTP/1.1'
The version string to write in the Status-Line of all HTTP responses, for example, "HTTP/1.1" (the default). Depending on the HTTP server used, this should also limit the supported features used in the response.

shutdown_timeout = 5
The time to wait for HTTP worker threads to clean up.

socket_file = None
If given, the name of the UNIX socket to use instead of TCP/IP.

When this option is not None, the *socket_host* and *socket_port* options are ignored.

property socket_host
The hostname or IP address on which to listen for connections.

Host values may be any IPv4 or IPv6 address, or any valid hostname. The string 'localhost' is a synonym for '127.0.0.1' (or '::1', if your hosts file prefers IPv6). The string '0.0.0.0' is a special IPv4 entry meaning "any active interface" (INADDR_ANY), and '::' is the similar IN6ADDR_ANY for IPv6. The empty string or None are not allowed.

socket_port = 8080
The TCP port on which to listen for connections.

socket_queue_size = 5
The 'backlog' argument to socket.listen(); specifies the maximum number of queued connections (default 5).

socket_timeout = 10
The timeout in seconds for accepted connections (default 10).

ssl_certificate = None
The filename of the SSL certificate to use.

ssl_certificate_chain = None
When using PyOpenSSL, the certificate chain to pass to Context.load_verify_locations.

ssl_ciphers = None
The ciphers list of SSL.

ssl_context = None
When using PyOpenSSL, an instance of SSL.Context.

ssl_module = 'builtin'
The name of a registered SSL adaptation module to use with the builtin WSGI server. Builtin options are: 'builtin' (to use the SSL library built into recent versions of Python). You may also register your own classes in the cheroot.server.ssl_adapters dict.

ssl_private_key = None
The filename of the private key to use with SSL.

start()
Start the HTTP server.

statistics = False
Turns statistics-gathering on or off for aware HTTP servers.

thread_pool = 10
The number of worker threads to start up in the pool.

thread_pool_max = -1
The maximum size of the worker-thread pool. Use -1 to indicate no limit.

wsgi_version = (1, 0)
The WSGI version tuple to use with the builtin WSGI server. The provided options are (1, 0) [which includes support for PEP 3333, which declares it covers WSGI version 1.0.1 but still mandates the wsgi.version (1, 0)] and ('u', 0), an experimental unicode version. You may create and register your own experimental versions of the WSGI protocol by adding custom classes to the cheroot.server.wsgi_gateways dict.

cherrypy._cptools module

CherryPy tools. A "tool" is any helper, adapted to CP.

Tools are usually designed to be used in a variety of ways (although some may only offer one if they choose):

Library calls All tools are callables that can be used wherever needed. The arguments are straightforward and should be detailed within the docstring.

Function decorators All tools, when called, may be used as decorators which configure individual CherryPy page handlers (methods on the CherryPy tree). That is, "@tools.anytool()" should "turn on" the tool via the decorated function's _cp_config attribute.

CherryPy config If a tool exposes a "_setup" callable, it will be called once per Request (if the feature is "turned on" via config).

Tools may be implemented as any object with a namespace. The builtins are generally either modules or instances of the tools.Tool class.

class cherrypy._cptools.**CachingTool**(*point*, *callable*, *name=None*, *priority=50*)
Bases: *cherrypy._cptools.Tool*

Caching Tool for CherryPy.

_setup()
Hook caching into cherrypy.request.

_wrapper(***kwargs*)

class cherrypy._cptools.**ErrorTool**(*callable*, *name=None*)
Bases: *cherrypy._cptools.Tool*

Tool which is used to replace the default request.error_response.

_setup()
Hook this tool into cherrypy.request.

The standard CherryPy request object will automatically call this method when the tool is "turned on" in config.

_wrapper()

class cherrypy._cptools.**HandlerTool**(*callable*, *name=None*)
Bases: *cherrypy._cptools.Tool*

Tool which is called 'before main', that may skip normal handlers.

If the tool successfully handles the request (by setting response.body), if should return True. This will cause CherryPy to skip any 'normal' page handler. If the tool did not handle the request, it should return False to tell CherryPy to continue on and call the normal page handler. If the tool is declared AS a page handler (see the 'handler' method), returning False will raise NotFound.

_setup()
Hook this tool into cherrypy.request.

The standard CherryPy request object will automatically call this method when the tool is "turned on" in config.

_wrapper(***kwargs*)

handler(**args*, ***kwargs*)
Use this tool as a CherryPy page handler.

For example:

```
class Root:
    nav = tools.staticdir.handler(section="/nav", dir="nav",
                                  root=absDir)
```

class cherrypy._cptools.**HandlerWrapperTool**(*newhandler*, *point='before_handler'*, *name=None*, *priority=50*)

Bases: *cherrypy._cptools.Tool*

Tool which wraps request.handler in a provided wrapper function.

The 'newhandler' arg must be a handler wrapper function that takes a 'next_handler' argument, plus `*args` and `**kwargs`. Like all page handler functions, it must return an iterable for use as cherrypy.response.body.

For example, to allow your 'inner' page handlers to return dicts which then get interpolated into a template:

```
def interpolator(next_handler, *args, **kwargs):
    filename = cherrypy.request.config.get('template')
    cherrypy.response.template = env.get_template(filename)
    response_dict = next_handler(*args, **kwargs)
    return cherrypy.response.template.render(**response_dict)
cherrypy.tools.jinja = HandlerWrapperTool(interpolator)
```

callable(**args*, ***kwargs*)

class cherrypy._cptools.**SessionAuthTool**(*callable*, *name=None*)

Bases: *cherrypy._cptools.HandlerTool*

class cherrypy._cptools.**SessionTool**

Bases: *cherrypy._cptools.Tool*

Session Tool for CherryPy.

sessions.locking When 'implicit' (the default), the session will be locked for you, just before running the page handler.

When 'early', the session will be locked before reading the request body. This is off by default for safety reasons; for example, a large upload would block the session, denying an AJAX progress meter (issue).

When 'explicit' (or any other value), you need to call cherrypy.session.acquire_lock() yourself before using session data.

_lock_session()

_setup()

Hook this tool into cherrypy.request.

The standard CherryPy request object will automatically call this method when the tool is "turned on" in config.

regenerate()

Drop the current session and make a new one (with a new id).

class cherrypy._cptools.**Tool**(*point*, *callable*, *name=None*, *priority=50*)

Bases: object

A registered function for use with CherryPy request-processing hooks.

help(tool.callable) should give you more information about this Tool.

_merged_args(*d=None*)

Return a dict of configuration entries for this Tool.

_setargs()
Copy func parameter names to obj attributes.

_setup()
Hook this tool into cherrypy.request.

The standard CherryPy request object will automatically call this method when the tool is "turned on" in config.

namespace = 'tools'

property on

class cherrypy._cptools.**Toolbox**(*namespace*)
Bases: object

A collection of Tools.

This object also functions as a config namespace handler for itself. Custom toolboxes should be added to each Application's toolboxes dict.

register(*point*, ***kwargs*)
Return a decorator which registers the function at the given hook point.

class cherrypy._cptools.**XMLRPCController**
Bases: object

A Controller (page handler collection) for XML-RPC.

To use it, have your controllers subclass this base class (it will turn on the tool for you).

You can also supply the following optional config entries:

```
tools.xmlrpc.encoding: 'utf-8'
tools.xmlrpc.allow_none: 0
```

XML-RPC is a rather discontinuous layer over HTTP; dispatching to the appropriate handler must first be performed according to the URL, and then a second dispatch step must take place according to the RPC method specified in the request body. It also allows a superfluous "/RPC2" prefix in the URL, supplies its own handler args in the body, and requires a 200 OK "Fault" response instead of 404 when the desired method is not found.

Therefore, XML-RPC cannot be implemented for CherryPy via a Tool alone. This Controller acts as the dispatch target for the first half (based on the URL); it then reads the RPC method from the request body and does its own second dispatch step based on that method. It also reads body params, and returns a Fault on error.

The XMLRPCDispatcher strips any /RPC2 prefix; if you aren't using /RPC2 in your URL's, you can safely skip turning on the XMLRPCDispatcher. Otherwise, you need to use declare it in config:

```
request.dispatch: cherrypy.dispatch.XMLRPCDispatcher()
```

_cp_config = {'tools.xmlrpc.on': True}

default(**vpath*, ***params*)

cherrypy._cptools.**_getargs**(*func*)
Return the names of all static arguments to the given function.

cherrypy._cptree module

CherryPy Application and Tree objects.

class cherrypy._cptree.**Application**(*root*, *script_name=''*, *config=None*)
Bases: object

A CherryPy Application.

Servers and gateways should not instantiate Request objects directly. Instead, they should ask an Application object for a request object.

An instance of this class may also be used as a WSGI callable (WSGI application object) for itself.

config = {}
pathconf} pairs, where 'pathconf' is itself a dict of {key: value} pairs.

Type A dict of {path

find_config(*path*, *key*, *default=None*)
Return the most-specific value for key along path, or default.

get_serving(*local*, *remote*, *scheme*, *sproto*)
Create and return a Request and Response object.

log = None
A LogManager instance. See _cplogging.

merge(*config*)
Merge the given config into self.config.

namespaces = {}

relative_urls = False

release_serving()
Release the current serving (request and response).

request_class
alias of *cherrypy._cprequest.Request*

response_class
alias of *cherrypy._cprequest.Response*

root = None
The top-most container of page handlers for this app. Handlers should be arranged in a hierarchy of attributes, matching the expected URI hierarchy; the default dispatcher then searches this hierarchy for a matching handler. When using a dispatcher other than the default, this value may be None.

property script_name
The URI "mount point" for this app.

A mount point is that portion of the URI which is constant for all URIs that are serviced by this application; it does not include scheme, host, or proxy ("virtual host") portions of the URI.

For example, if script_name is "/my/cool/app", then the URL "http://www.example.com/my/cool/app/page1" might be handled by a "page1" method on the root object.

The value of script_name MUST NOT end in a slash. If the script_name refers to the root of the URI, it MUST be an empty string (not "/").

If script_name is explicitly set to None, then the script_name will be provided for each call from request.wsgi_environ['SCRIPT_NAME'].

script_name_doc = 'The URI "mount point" for this app. A mount point\n is that portion

toolboxes = {'tools': <cherrypy._cptools.Toolbox object>}

wsgiapp = None
A CPWSGIApp instance. See _cpwsgi.

class cherrypy._cptree.**Tree**
Bases: object

A registry of CherryPy applications, mounted at diverse points.

An instance of this class may also be used as a WSGI callable (WSGI application object), in which case it dispatches to all mounted apps.

apps = {}
application}, where "script name" is a string declaring the URI mount point (no trailing slash), and "application" is an instance of cherrypy.Application (or an arbitrary WSGI callable if you happen to be using a WSGI server).

Type A dict of the form {script name

graft(*wsgi_callable*, *script_name=''*)
Mount a wsgi callable at the given script_name.

mount(*root*, *script_name=''*, *config=None*)
Mount a new app from a root object, script_name, and config.

root An instance of a "controller class" (a collection of page handler methods) which represents the root of the application. This may also be an Application instance, or None if using a dispatcher other than the default.

script_name A string containing the "mount point" of the application. This should start with a slash, and be the path portion of the URL at which to mount the given root. For example, if root.index() will handle requests to "http://www.example.com:8080/dept/app1/", then the script_name argument would be "/dept/app1".

It MUST NOT end in a slash. If the script_name refers to the root of the URI, it MUST be an empty string (not "/").

config A file or dict containing application config.

script_name(*path=None*)
Return the script_name of the app at the given path, or None.

If path is None, cherrypy.request is used.

cherrypy._cpwsgi module

WSGI interface (see PEP 333 and 3333).

Note that WSGI environ keys and values are 'native strings'; that is, whatever the type of "" is. For Python 2, that's a byte string; for Python 3, it's a unicode string. But PEP 3333 says: "even if Python's str type is actually Unicode "under the hood", the content of native strings must still be translatable to bytes via the Latin-1 encoding!"

class cherrypy._cpwsgi.**AppResponse**(*environ*, *start_response*, *cpapp*)
Bases: object

WSGI response iterable for CherryPy applications.

close()
Close and de-reference the current request and response. (Core)

headerNames = {'CONTENT_LENGTH': 'Content-Length', 'CONTENT_TYPE': 'Content-Type', 'HT

recode_path_qs(*path*, *qs*)

run()
Create a Request object using environ.

translate_headers(*environ*)
Translate CGI-environ header names to HTTP header names.

class cherrypy._cpwsgi.**CPWSGIApp**(*cpapp*, *pipeline=None*)
Bases: object

A WSGI application object for a CherryPy Application.

config = {}
A dict whose keys match names listed in the pipeline. Each value is a further dict which will be passed to the corresponding named WSGI callable (from the pipeline) as keyword arguments.

head = None
Rather than nest all apps in the pipeline on each call, it's only done the first time, and the result is memoized into self.head. Set this to None again if you change self.pipeline after calling self.

namespace_handler(*k*, *v*)
Config handler for the 'wsgi' namespace.

pipeline = [('ExceptionTrapper', <class 'cherrypy._cpwsgi.ExceptionTrapper'>), ('Inter
A list of (name, wsgiapp) pairs. Each 'wsgiapp' MUST be a constructor that takes an initial, positional 'nextapp' argument, plus optional keyword arguments, and returns a WSGI application (that takes environ and start_response arguments). The 'name' can be any you choose, and will correspond to keys in self.config.

response_class
The class to instantiate and return as the next app in the WSGI chain.

alias of *cherrypy._cpwsgi.AppResponse*

tail(*environ*, *start_response*)
WSGI application callable for the actual CherryPy application.

You probably shouldn't call this; call self.__call__ instead, so that any WSGI middleware in self.pipeline can run first.

class cherrypy._cpwsgi.**ExceptionTrapper**(*nextapp*, *throws=(<class 'KeyboardInterrupt'>, <class 'SystemExit'>)*)
Bases: object

WSGI middleware that traps exceptions.

class cherrypy._cpwsgi.**InternalRedirector**(*nextapp*, *recursive=False*)
Bases: object

WSGI middleware that handles raised cherrypy.InternalRedirect.

class cherrypy._cpwsgi.**VirtualHost**(*default*, *domains=None*, *use_x_forwarded_host=True*)
Bases: object

Select a different WSGI application based on the Host header.

This can be useful when running multiple sites within one CP server. It allows several domains to point to different applications. For example:

```
root = Root()
RootApp = cherrypy.Application(root)
Domain2App = cherrypy.Application(root)
```

(continues on next page)

(continued from previous page)

```
SecureApp = cherrypy.Application(Secure())

vhost = cherrypy._cpwsgi.VirtualHost(
    RootApp,
    domains={
        'www.domain2.example': Domain2App,
        'www.domain2.example:443': SecureApp,
    },
)

cherrypy.tree.graft(vhost)
```

default = None
Required. The default WSGI application.

domains = {}
application} pairs. The incoming "Host" request header is looked up in this dict, and, if a match is found, the corresponding WSGI application will be called instead of the default. Note that you often need separate entries for "example.com" and "www.example.com". In addition, "Host" headers may contain the port number.

Type A dict of {host header value

use_x_forwarded_host = True
If True (the default), any "X-Forwarded-Host" request header will be used instead of the "Host" header. This is commonly added by HTTP servers (such as Apache) when proxying.

class cherrypy._cpwsgi.**_TrappedResponse**(*nextapp*, *environ*, *start_response*, *throws*)
Bases: object

close()

response = <list_iterator object>

trap(*func*, **args*, ***kwargs*)

cherrypy._cpwsgi.**downgrade_wsgi_ux_to_1x**(*environ*)
Return a new environ dict for WSGI 1.x from the given WSGI u.x environ.

cherrypy._cpwsgi_server module

WSGI server interface (see PEP 333).

This adds some CP-specific bits to the framework-agnostic cheroot package.

class cherrypy._cpwsgi_server.**CPWSGIHTTPRequest**(*server*, *conn*)
Bases: cheroot.server.HTTPRequest

Wrapper for cheroot.server.HTTPRequest.

This is a layer, which preserves URI parsing mode like it which was before Cheroot v5.8.0.

class cherrypy._cpwsgi_server.**CPWSGIServer**(*server_adapter=<cherrypy._cpserver.Server object>*)
Bases: cheroot.wsgi.Server

Wrapper for cheroot.wsgi.Server.

cheroot has been designed to not reference CherryPy in any way, so that it can be used in other frameworks and applications. Therefore, we wrap it here, so we can set our own mount points from cherrypy.tree and apply some attributes from config -> cherrypy.server -> wsgi.Server.

error_log(*msg=''*, *level=20*, *traceback=False*)
Write given message to the error log.

fmt = 'CherryPy/{cherrypy.__version__} {cheroot.wsgi.Server.version}'

version = 'CherryPy/18.6.1.dev49+g98929b51 Cheroot/8.5.1'

cherrypy._helper module

Helper functions for CP apps.

class cherrypy._helper.**_ClassPropertyDescriptor**(*fget*, *fset=None*)
Bases: object

Descript for read-only class-based property.

Turns a classmethod-decorated func into a read-only property of that class type (means the value cannot be set).

cherrypy._helper.**classproperty**(*func*)
Decorator like classmethod to implement a static class property.

cherrypy._helper.**expose**(*func=None*, *alias=None*)
Expose the function or class.

Optionally provide an alias or set of aliases.

cherrypy._helper.**normalize_path**(*path*)
Resolve given path from relative into absolute form.

cherrypy._helper.**popargs**(**args*, ***kwargs*)
Decorate _cp_dispatch.

(cherrypy.dispatch.Dispatcher.dispatch_method_name)

Optional keyword argument: handler=(Object or Function)

Provides a _cp_dispatch function that pops off path segments into cherrypy.request.params under the names specified. The dispatch is then forwarded on to the next vpath element.

Note that any existing (and exposed) member function of the class that popargs is applied to will override that value of the argument. For instance, if you have a method named "list" on the class decorated with popargs, then accessing "/list" will call that function instead of popping it off as the requested parameter. This restriction applies to all _cp_dispatch functions. The only way around this restriction is to create a "blank class" whose only function is to provide _cp_dispatch.

If there are path elements after the arguments, or more arguments are requested than are available in the vpath, then the 'handler' keyword argument specifies the next object to handle the parameterized request. If handler is not specified or is None, then self is used. If handler is a function rather than an instance, then that function will be called with the args specified and the return value from that function used as the next object INSTEAD of adding the parameters to cherrypy.request.args.

This decorator may be used in one of two ways:

As a class decorator:

```
@cherrypy.popargs('year', 'month', 'day')
class Blog:
    def index(self, year=None, month=None, day=None):
        #Process the parameters here; any url like
        #/, /2009, /2009/12, or /2009/12/31
        #will fill in the appropriate parameters.
```

(continues on next page)

(continued from previous page)

```
    def create(self):
        #This link will still be available at /create.
        #Defined functions take precedence over arguments.
```

Or as a member of a class:

```
class Blog:
    _cp_dispatch = cherrypy.popargs('year', 'month', 'day')
    #...
```

The handler argument may be used to mix arguments with built in functions. For instance, the following setup allows different activities at the day, month, and year level:

```
class DayHandler:
    def index(self, year, month, day):
        #Do something with this day; probably list entries

    def delete(self, year, month, day):
        #Delete all entries for this day

@cherrypy.popargs('day', handler=DayHandler())
class MonthHandler:
    def index(self, year, month):
        #Do something with this month; probably list entries

    def delete(self, year, month):
        #Delete all entries for this month

@cherrypy.popargs('month', handler=MonthHandler())
class YearHandler:
    def index(self, year):
        #Do something with this year

    #...

@cherrypy.popargs('year', handler=YearHandler())
class Root:
    def index(self):
        #...
```

cherrypy._helper.**url**(*path=''*, *qs=''*, *script_name=None*, *base=None*, *relative=None*)

Create an absolute URL for the given path.

If 'path' starts with a slash ('/'), this will return (base + script_name + path + qs).

If it does not start with a slash, this returns (base + script_name [+ request.path_info] + path + qs).

If script_name is None, cherrypy.request will be used to find a script_name, if available.

If base is None, cherrypy.request.base will be used (if available). Note that you can use cherrypy.tools.proxy to change this.

Finally, note that this function can be used to obtain an absolute URL for the current request path (minus the querystring) by passing no args. If you call url(qs=cherrypy.request.query_string), you should get the original browser URL (assuming no internal redirections).

If relative is None or not provided, request.app.relative_urls will be used (if available, else False). If False, the output will be an absolute URL (including the scheme, host, vhost, and script_name). If True, the output will

instead be a URL that is relative to the current request path, perhaps including '..' atoms. If relative is the string 'server', the output will instead be a URL that is relative to the server root; i.e., it will start with a slash.

cherrypy._json module

JSON support.

Expose preferred json module as json and provide encode/decode convenience functions.

`cherrypy._json.`**`decode`**(*s*, *_w=<built-in method match of _sre.SRE_Pattern object>*)
Return the Python representation of `s` (a `str` instance containing a JSON document).

`cherrypy._json.`**`encode`**(*value*)
Encode to bytes.

cherrypy.daemon module

The CherryPy daemon.

`cherrypy.daemon.`**`run`**()
Run cherryd CLI.

`cherrypy.daemon.`**`start`**(*configfiles=None*, *daemonize=False*, *environment=None*, *fastcgi=False*, *scgi=False*, *pidfile=None*, *imports=None*, *cgi=False*)
Subscribe all engine plugins and start the engine.

15.1.3 Module contents

CherryPy is a pythonic, object-oriented HTTP framework.

CherryPy consists of not one, but four separate API layers.

The APPLICATION LAYER is the simplest. CherryPy applications are written as a tree of classes and methods, where each branch in the tree corresponds to a branch in the URL path. Each method is a 'page handler', which receives GET and POST params as keyword arguments, and returns or yields the (HTML) body of the response. The special method name 'index' is used for paths that end in a slash, and the special method name 'default' is used to handle multiple paths via a single handler. This layer also includes:

- the 'exposed' attribute (and cherrypy.expose)
- cherrypy.quickstart()
- _cp_config attributes
- cherrypy.tools (including cherrypy.session)
- cherrypy.url()

The ENVIRONMENT LAYER is used by developers at all levels. It provides information about the current request and response, plus the application and server environment, via a (default) set of top-level objects:

- cherrypy.request
- cherrypy.response
- cherrypy.engine
- cherrypy.server
- cherrypy.tree

- cherrypy.config
- cherrypy.thread_data
- cherrypy.log
- cherrypy.HTTPError, NotFound, and HTTPRedirect
- cherrypy.lib

The EXTENSION LAYER allows advanced users to construct and share their own plugins. It consists of:

- Hook API
- Tool API
- Toolbox API
- Dispatch API
- Config Namespace API

Finally, there is the CORE LAYER, which uses the core API's to construct the default components which are available at higher layers. You can think of the default components as the 'reference implementation' for CherryPy. Megaframeworks (and advanced users) may replace the default components with customized or extended components. The core API's are:

- Application API
- Engine API
- Request API
- Server API
- WSGI API

These API's are described in the CherryPy specification.

class cherrypy.**Application**(*root*, *script_name=''*, *config=None*)
Bases: object

A CherryPy Application.

Servers and gateways should not instantiate Request objects directly. Instead, they should ask an Application object for a request object.

An instance of this class may also be used as a WSGI callable (WSGI application object) for itself.

config = {}
pathconf} pairs, where 'pathconf' is itself a dict of {key: value} pairs.

Type A dict of {path

find_config(*path*, *key*, *default=None*)
Return the most-specific value for key along path, or default.

get_serving(*local*, *remote*, *scheme*, *sproto*)
Create and return a Request and Response object.

log = None
A LogManager instance. See _cplogging.

merge(*config*)
Merge the given config into self.config.

namespaces = {}

relative_urls = False

release_serving()
Release the current serving (request and response).

request_class
alias of *cherrypy._cprequest.Request*

response_class
alias of *cherrypy._cprequest.Response*

root = None
The top-most container of page handlers for this app. Handlers should be arranged in a hierarchy of attributes, matching the expected URI hierarchy; the default dispatcher then searches this hierarchy for a matching handler. When using a dispatcher other than the default, this value may be None.

property script_name
The URI "mount point" for this app.

A mount point is that portion of the URI which is constant for all URIs that are serviced by this application; it does not include scheme, host, or proxy ("virtual host") portions of the URI.

For example, if script_name is "/my/cool/app", then the URL "http://www.example.com/my/cool/app/page1" might be handled by a "page1" method on the root object.

The value of script_name MUST NOT end in a slash. If the script_name refers to the root of the URI, it MUST be an empty string (not "/").

If script_name is explicitly set to None, then the script_name will be provided for each call from request.wsgi_environ['SCRIPT_NAME'].

script_name_doc = 'The URI "mount point" for this app. A mount point\n is that portion

toolboxes = {'tools': <cherrypy._cptools.Toolbox object>}

wsgiapp = None
A CPWSGIApp instance. See _cpwsgi.

exception cherrypy.**CherryPyException**
Bases: Exception

A base class for CherryPy exceptions.

exception cherrypy.**HTTPError**(*status=500*, *message=None*)
Bases: *cherrypy._cperror.CherryPyException*

Exception used to return an HTTP error code (4xx-5xx) to the client.

This exception can be used to automatically send a response using a http status code, with an appropriate error page. It takes an optional `status` argument (which must be between 400 and 599); it defaults to 500 ("Internal Server Error"). It also takes an optional `message` argument, which will be returned in the response body. See RFC2616 for a complete list of available error codes and when to use them.

Examples:

```
raise cherrypy.HTTPError(403)
raise cherrypy.HTTPError(
    "403 Forbidden", "You are not allowed to access this resource.")
```

code = None
The integer HTTP status code.

get_error_page(**args*, ***kwargs*)

classmethod handle(*exception*, *status=500*, *message=''*)
Translate exception into an HTTPError.

reason = None
The HTTP Reason-Phrase string.

set_response()
Modify cherrypy.response status, headers, and body to represent self.

CherryPy uses this internally, but you can also use it to create an HTTPError object and set its output without *raising* the exception.

status = None
The HTTP status code. May be of type int or str (with a Reason-Phrase).

exception cherrypy.**HTTPRedirect**(*urls*, *status=None*, *encoding=None*)
Bases: *cherrypy._cperror.CherryPyException*

Exception raised when the request should be redirected.

This exception will force a HTTP redirect to the URL or URL's you give it. The new URL must be passed as the first argument to the Exception, e.g., HTTPRedirect(newUrl). Multiple URLs are allowed in a list. If a URL is absolute, it will be used as-is. If it is relative, it is assumed to be relative to the current cherrypy.request.path_info.

If one of the provided URL is a unicode object, it will be encoded using the default encoding or the one passed in parameter.

There are multiple types of redirect, from which you can select via the `status` argument. If you do not provide a `status` arg, it defaults to 303 (or 302 if responding with HTTP/1.0).

Examples:

```
raise cherrypy.HTTPRedirect("")
raise cherrypy.HTTPRedirect("/abs/path", 307)
raise cherrypy.HTTPRedirect(["path1", "path2?a=1&b=2"], 301)
```

See *Redirecting POST* for additional caveats.

default_status = 303

encoding = 'utf-8'
The encoding when passed urls are not native strings

set_response()
Modify cherrypy.response status, headers, and body to represent self.

CherryPy uses this internally, but you can also use it to create an HTTPRedirect object and set its output without *raising* the exception.

property status
The integer HTTP status code to emit.

urls = None
The list of URL's to emit.

exception cherrypy.**InternalRedirect**(*path*, *query_string=''*)
Bases: *cherrypy._cperror.CherryPyException*

Exception raised to switch to the handler for a different URL.

This exception will redirect processing to another path within the site (without informing the client). Provide the new path as an argument when raising the exception. Provide any params in the querystring for the new URL.

exception cherrypy.**NotFound**(*path=None*)
Bases: *cherrypy._cperror.HTTPError*

Exception raised when a URL could not be mapped to any handler (404).

This is equivalent to raising *HTTPError("404 Not Found")*.

class cherrypy.**Tool**(*point*, *callable*, *name=None*, *priority=50*)
Bases: object

A registered function for use with CherryPy request-processing hooks.

help(tool.callable) should give you more information about this Tool.

_merged_args(*d=None*)
Return a dict of configuration entries for this Tool.

_setargs()
Copy func parameter names to obj attributes.

_setup()
Hook this tool into cherrypy.request.

The standard CherryPy request object will automatically call this method when the tool is "turned on" in config.

namespace = 'tools'

property on

cherrypy.**expose**(*func=None*, *alias=None*)
Expose the function or class.

Optionally provide an alias or set of aliases.

cherrypy.**popargs**(**args*, ***kwargs*)
Decorate _cp_dispatch.

(cherrypy.dispatch.Dispatcher.dispatch_method_name)

Optional keyword argument: handler=(Object or Function)

Provides a _cp_dispatch function that pops off path segments into cherrypy.request.params under the names specified. The dispatch is then forwarded on to the next vpath element.

Note that any existing (and exposed) member function of the class that popargs is applied to will override that value of the argument. For instance, if you have a method named "list" on the class decorated with popargs, then accessing "/list" will call that function instead of popping it off as the requested parameter. This restriction applies to all _cp_dispatch functions. The only way around this restriction is to create a "blank class" whose only function is to provide _cp_dispatch.

If there are path elements after the arguments, or more arguments are requested than are available in the vpath, then the 'handler' keyword argument specifies the next object to handle the parameterized request. If handler is not specified or is None, then self is used. If handler is a function rather than an instance, then that function will be called with the args specified and the return value from that function used as the next object INSTEAD of adding the parameters to cherrypy.request.args.

This decorator may be used in one of two ways:

As a class decorator:

```
@cherrypy.popargs('year', 'month', 'day')
class Blog:
    def index(self, year=None, month=None, day=None):
```

(continues on next page)

(continued from previous page)

```
        #Process the parameters here; any url like
        #/, /2009, /2009/12, or /2009/12/31
        #will fill in the appropriate parameters.

    def create(self):
        #This link will still be available at /create.
        #Defined functions take precedence over arguments.
```

Or as a member of a class:

```
class Blog:
    _cp_dispatch = cherrypy.popargs('year', 'month', 'day')
    #...
```

The handler argument may be used to mix arguments with built in functions. For instance, the following setup allows different activities at the day, month, and year level:

```
class DayHandler:
    def index(self, year, month, day):
        #Do something with this day; probably list entries

    def delete(self, year, month, day):
        #Delete all entries for this day

@cherrypy.popargs('day', handler=DayHandler())
class MonthHandler:
    def index(self, year, month):
        #Do something with this month; probably list entries

    def delete(self, year, month):
        #Delete all entries for this month

@cherrypy.popargs('month', handler=MonthHandler())
class YearHandler:
    def index(self, year):
        #Do something with this year

    #...

@cherrypy.popargs('year', handler=YearHandler())
class Root:
    def index(self):
        #...
```

cherrypy.**quickstart**(*root=None*, *script_name=''*, *config=None*)

Mount the given root, start the builtin server (and engine), then block.

root: an instance of a "controller class" (a collection of page handler methods) which represents the root of the application.

script_name: a string containing the "mount point" of the application. This should start with a slash, and be the path portion of the URL at which to mount the given root. For example, if root.index() will handle requests to "http://www.example.com:8080/dept/app1/", then the script_name argument would be "/dept/app1".

It MUST NOT end in a slash. If the script_name refers to the root of the URI, it MUST be an empty string (not "/").

config: a file or dict containing application config. If this contains a [global] section, those entries will be used in the global (site-wide) config.

cherrypy.**url**(*path=''*, *qs=''*, *script_name=None*, *base=None*, *relative=None*)

Create an absolute URL for the given path.

If 'path' starts with a slash ('/'), this will return (base + script_name + path + qs).

If it does not start with a slash, this returns (base + script_name [+ request.path_info] + path + qs).

If script_name is None, cherrypy.request will be used to find a script_name, if available.

If base is None, cherrypy.request.base will be used (if available). Note that you can use cherrypy.tools.proxy to change this.

Finally, note that this function can be used to obtain an absolute URL for the current request path (minus the querystring) by passing no args. If you call url(qs=cherrypy.request.query_string), you should get the original browser URL (assuming no internal redirections).

If relative is None or not provided, request.app.relative_urls will be used (if available, else False). If False, the output will be an absolute URL (including the scheme, host, vhost, and script_name). If True, the output will instead be a URL that is relative to the current request path, perhaps including '..' atoms. If relative is the string 'server', the output will instead be a URL that is relative to the server root; i.e., it will start with a slash.

CherryPy is a pythonic, object-oriented web framework.

CherryPy allows developers to build web applications in much the same way they would build any other object-oriented Python program. This results in smaller source code developed in less time.

CherryPy is now more than ten years old and it is has proven to be fast and reliable. It is being used in production by many sites, from the simplest to the most demanding.

A CherryPy application typically looks like this:

```
import cherrypy

class HelloWorld(object):
    @cherrypy.expose
    def index(self):
        return "Hello World!"

cherrypy.quickstart(HelloWorld())
```

In order to make the most of CherryPy, you should start with the *tutorials* that will lead you through the most common aspects of the framework. Once done, you will probably want to browse through the *basics* and *advanced* sections that will demonstrate how to implement certain operations. Finally, you will want to carefully read the configuration and *extend* sections that go in-depth regarding the powerful features provided by the framework.

Above all, have fun with your application!

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X