CherryPy Documentation

Release 3.2.4

CherryPy Team

Jun 30, 2017

Contents

CHAPTER 1

Foreword

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 IRC (#cherrypy on OFTC). 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!

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 IRC (#cherrypy on OFTC).

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.

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.

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 2

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

Requirements

CherryPy does not have any mandatory requirements. However certain features it comes with will require you install certain packages.

- routes for declarative URL mapping dispatcher
- psycopg2 for PostgreSQL backend session
- pywin32 for Windows services

- python-memcached for memcached backend session
- simplejson for a better JSON support
- pyOpenSSL if your Python environment does not have the builtin `ssl` module

Supported python version

CherryPy supports Python 2.3 through to 3.4.

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 BitBucket:

```
$ hg clone https://bitbucket.org/cherrypy/cherrypy
$ cd cherrypy
$ python setup.py install
```

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 Started monitor thread '_TimeoutMonitor'.
[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.

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.

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 3

Tutorials

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

Contents

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 Started monitor thread '_TimeoutMonitor'.
[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.

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

class StringGenerator(object):
```

```
    @cherrypy.expose
    def index(self):
        return "Hello world!"

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

if __name__ == '__main__':
    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*

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 not assume you wish to indicate the length of that string dynamically.

```
import random
import string

import cherrypy

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

    @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 *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/password 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 and 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.

Tutorial 4: Submit this form

CherryPy is a web framework upon which you build web applications. The most traditionnal 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.

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.

```
1  import random
2  import string
3
4  import cherrypy
5
6  class StringGenerator(object):
7      @cherrypy.expose
8      def index(self):
9          return """<html>
10           <head></head>
11       <body>
12           <form method="get" action="generate">
13             <input type="text" value="8" name="length" />
14                 <button type="submit">Give it now!</button>
15           </form>
16       </body>
17     </html>"""
18
19     @cherrypy.expose
20     def generate(self, length=8):
21         some_string = ''.join(random.sample(string.hexdigits, int(length)))
22         cherrypy.session['mystring'] = some_string
23         return some_string
24
25     @cherrypy.expose
26     def display(self):
27         return cherrypy.session['mystring']
28
29 if __name__ == '__main__':
30      conf = {
31          '/': {
32              'tools.sessions.on': True
33          }
34      }
35      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://localhostt:8080/, generate a random string, then go to http://localhostt: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.

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

Web application 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*.

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

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

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

import cherrypy

class StringGenerator(object):
   @cherrypy.expose
   def index(self):
       return """<html>
         <head>
           <link href="/static/css/style.css" rel="stylesheet">
         </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,
            'tools.staticdir.root': os.path.abspath(os.getcwd())
        },
        '/static': {
            'tools.staticdir.on': True,
            'tools.staticdir.dir': './public'
        }
    }
```

```
43      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 non-absolute 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*.

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 midly following REST principles.

```
1   import random
2   import string
3
4   import cherrypy
5
6   class StringGeneratorWebService(object):
7       exposed = True
8
9       @cherrypy.tools.accept(media='text/plain')
10      def GET(self):
11          return cherrypy.session['mystring']
12
13      def POST(self, length=8):
14          some_string = ''.join(random.sample(string.hexdigits, int(length)))
15          cherrypy.session['mystring'] = some_string
16          return some_string
17
18      def PUT(self, another_string):
19          cherrypy.session['mystring'] = another_string
```

```
20
21      def DELETE(self):
22          cherrypy.session.pop('mystring', None)
23
24  if __name__ == '__main__':
25      conf = {
26          '/': {
27              'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
28              'tools.sessions.on': True,
29              'tools.response_headers.on': True,
30              'tools.response_headers.headers': [('Content-Type', 'text/plain')],
31          }
32      }
33      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 math 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 through the *exposed = True* attribute of the class itself.

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 oher 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 otherwiser.

Please install requests through the following command:

```
$ pip install requests
```

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

```
1  >>> import requests
2  >>> s = requests.Session()
3  >>> r = s.get('http://127.0.0.1:8080/')
4  >>> r.status_code
5  500
6  >>> r = s.post('http://127.0.0.1:8080/')
7  >>> r.status_code, r.text
8  (200, u'04A92138')
9  >>> r = s.get('http://127.0.0.1:8080/')
10 >>> r.status_code, r.text
11 (200, u'04A92138')
12 >>> r = s.get('http://127.0.0.1:8080/', headers={'Accept': 'application/json'})
13 >>> r.status_code
14 406
15 >>> r = s.put('http://127.0.0.1:8080/', params={'another_string': 'hello'})
16 >>> r = s.get('http://127.0.0.1:8080/')
17 >>> r.status_code, r.text
18 (200, u'hello')
19 >>> 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 steam 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 http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.7*

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.

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;
}

#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").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">
        <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 file('index.html')

class StringGeneratorWebService(object):
    exposed = True

    @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')],
        },
        '/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.

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 your 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 cherrypy

DB_STRING = "my.db"

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

class StringGeneratorWebService(object):
    exposed = True

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

    def POST(self, length=8):
        some_string = ''.join(random.sample(string.hexdigits, int(length)))
        with sqlite3.connect(DB_STRING) as c:
            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:
            c.execute("UPDATE user_string SET value=? WHERE session_id=?",
                      [another_string, cherrypy.session.id])

    def DELETE(self):
        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
    """
```

```
47      with sqlite3.connect(DB_STRING) as con:
48          con.execute("CREATE TABLE user_string (session_id, value)")
49
50  def cleanup_database():
51      """
52      Destroy the `user_string` table from the database
53      on server shutdown.
54      """
55      with sqlite3.connect(DB_STRING) as con:
56          con.execute("DROP TABLE user_string")
57
58  if __name__ == '__main__':
59      conf = {
60          '/': {
61              'tools.sessions.on': True,
62              'tools.staticdir.root': os.path.abspath(os.getcwd())
63          },
64          '/generator': {
65              'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
66              'tools.response_headers.on': True,
67              'tools.response_headers.headers': [('Content-Type', 'text/plain')],
68          },
69          '/static': {
70              'tools.staticdir.on': True,
71              'tools.staticdir.dir': './public'
72          }
73      }
74
75      cherrypy.engine.subscribe('start', setup_database)
76      cherrypy.engine.subscribe('stop', cleanup_database)
77
78      webapp = StringGenerator()
79      webapp.generator = StringGeneratorWebService()
80      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 76-77, 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.

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.

Tutorial 10: 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 your structure your application in a way that mirrors these high-level ideas.

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.

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.

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.

CHAPTER 4

Basics

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

Contents

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.

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.

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': ... }*

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*.

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:
    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.

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*.

Play along with your other loggers

Your application may obviously already use the `logging` module to trace application level messages. CherryPy will not interfere with them as long as your loggers are explicitely named. This would work nicely:

```
import logging
logger = logging.getLogger('myapp.mypackage')
logger.setLevel(logging.INFO)
stream = logging.StreamHandler()
stream.setLevel(logging.INFO)
logger.addHandler(stream)
```

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.

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.

Per-application configuration

To configure your application, pass in a dictionary or a file when you associate their 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.

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")
```

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):
        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')
```

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 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.

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_type = "file"
tools.sessions.storage_path = "/some/directorys"
```

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.

```
[/]
tools.sessions.on: True
tools.sessions.storage_type = "memcached"
```

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'
```

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*.

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"
```

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).

Dealing with JSON

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

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.

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.

Authentication

CherryPy provides support for two very simple authentication mechanisms, both described in **RFC 2617**: Basic and Digest. They are most commonly known to trigger a browser's popup asking users their name and password.

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

conf = {
   '/protected/area': {
       'tools.auth_basic.on': True,
       'tools.auth_basic.realm': 'localhost',
       'tools.auth_basic.checkpassword': validate_password
    }
}

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.

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'
   }
}

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

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 5

Advanced

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

Contents

- *WebSocket support*
- *Database support*
- *HTML Templating support*
- *Testing your application*

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.

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/<bandname>/
- http://hostname/<bandname>/albums/<recordname>/

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 explicitely declared in your source code. Luckily, CherryPy provides ways to support those use cases.

See also:

This section extends from this stackoverflow response.

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

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

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

@cherrypy.popargs('title')
class Album(object):
    @cherrypy.expose
    def index(self, name, title):
        return 'About %s by %s...' % (title, 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 name *band*
- take again the first segment (since we removed the previous first) and store it into a parameter named *title*

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

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

@cherrypy.popargs('num', 'track')
class Track(object):
    @cherrypy.expose
    def index(self, name, title, num, track):
        ...
```

This would handle the following URL:

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

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

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`.

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.

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.

Response timeouts

CherryPy responses include 3 attributes related to time:

- `response.time`: the `time.time()` at which the response began
- `response.timeout`: the number of seconds to allow responses to run
- `response.timed_out`: a boolean indicating whether the response has timed out (default False).

The request processing logic inspects the value of `response.timed_out` at various stages; if it is ever True, then `TimeoutError` is raised. You are free to do the same within your own code.

Rather than calculate the difference by hand, you can call `response.check_timeout` to set `timed_out` for you.

Note: The default response timeout is 300 seconds.

Timeout Monitor

In addition, CherryPy includes a `cherrypy.engine.timeout_monitor` which monitors all active requests in a separate thread; periodically, it calls `check_timeout` on them all. It is subscribed by default. To turn it off:

```
[global]
engine.timeout_monitor.on: False
```

or:

```
cherrypy.engine.timeout_monitor.unsubscribe()
```

You can also change the interval (in seconds) at which the timeout monitor runs:

```
[global]
engine.timeout_monitor.frequency: 60 * 60
```

The default is once per minute. The above example changes that to once per hour.

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()
```

Windows Console Events

Microsoft Windows uses console events to communicate some signals, like Ctrl-C. When deploying CherryPy on Windows platforms, you should obtain the Python for Windows Extensions; once you have them 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.

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

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'"

# set the priority according to your needs if you are hooking something
# else on the 'before_finalize' hook point.
cherrypy.tools.secureheaders = cherrypy.Tool('before_finalize', secureheaders,␣
↪priority=60)
```

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*.

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.

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

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.

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*.

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.

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.

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 here a recipe on how integrating SQLAlchemy using a mix of *plugins* and *tools*.

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*.

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')
```

```
    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.

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.

CHAPTER 6

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

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.

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.

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.

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.

Declaration

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

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()
```

_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:

```
def page(self):
    return "Hello, world!"
```

```
page.exposed = True
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")}

    def page(self):
        return "Hullo, Werld!"
    page.exposed = True
```

Note: This behavior is only guaranteed for the default dispatcher. Other dispatchers may have different restrictions on where you can attach `_cp_config` attributes.

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.

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".

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.

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**.

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]`.

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 7

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

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 accross 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).

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 passas as a parameter.

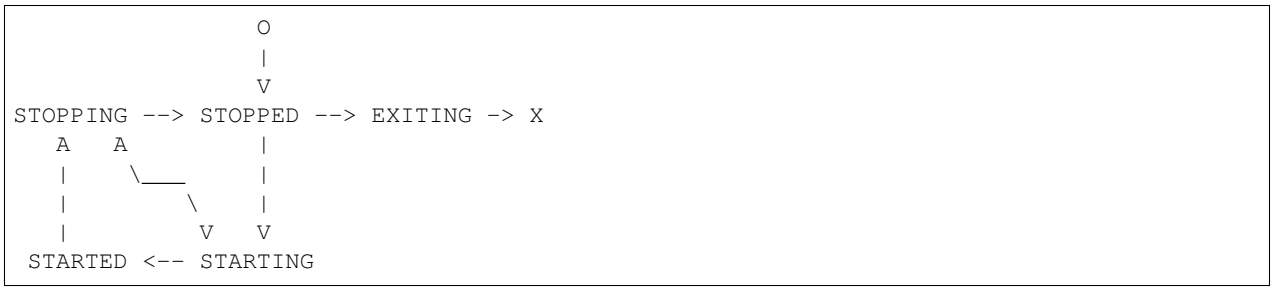
The whole behaviour happens synchronously and, in that sense, if a subscriber takes too long to process a message, the remaining subscribers will be delayed.

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.

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 accross 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/2014:13:04:07] ENGINE Started monitor thread '_TimeoutMonitor'.
[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"

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 Started monitor thread '_TimeoutMonitor'.
[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
```

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.

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.

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:

```
def log_it():
   print(cherrypy.request.remote.ip)

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

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()

    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.
def check_access(default=False):
    if not getattr(cherrypy.request, "userid", default):
        raise cherrypy.HTTPError(401)
newauthtools.check_access = cherrypy.Tool('before_request_body', check_access)
```

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)
```

Request parameters massaging

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
```

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 snipper, 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 2616#sec10.4.5**).

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 ther 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.

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 8

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

    * *Nginx*

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.

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()
```

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()
```

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.

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 algoritms 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'
    ```

    (a) *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
    ```

2. 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.

WSGI servers

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)

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()
```

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
```

uwsgi

You can use uwsgi HTTP server as follow:

```
import cherrypy

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

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
```

Reverse-proxying

Apache

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 9

Contribute

To be done.

CHAPTER 10

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

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!

Index