CherryPy

A Minimalist Python Web Framework

Contenu

[Introduction 3](#_Toc476139376)

[Homepage 4](#_Toc476139377)

[CherryPy-10.1.1-py2.py3-none-any.whl 5](#_Toc476139378)

[Utiliser Cherrypy (serveur web léger) avec Bottle 6](#_Toc476139379)

[Les Maths Libres 8](#_Toc476139380)

[Cherrypy 8](#_Toc476139381)

[29 avril 2016 CherryPy, pour qui, pour quoi ? 8](#_Toc476139382)

[Cherrypy, pour qui ? 8](#_Toc476139383)

[Cherrypy, pour quoi faire ? 8](#_Toc476139384)

[28 avril 2016 uploader un fichier 8](#_Toc476139385)

[L’idée 8](#_Toc476139386)

[Le code 9](#_Toc476139387)

[28 avril 2016 somme de deux entiers 10](#_Toc476139388)

[Une projet, deux solutions 10](#_Toc476139389)

[Première solution 10](#_Toc476139390)

[Deuxième solution 10](#_Toc476139391)

[27 avril 2016 Un début de configuration pour Cherrypy 11](#_Toc476139392)

[Une arborescence 11](#_Toc476139393)

[Le fichier server.conf 11](#_Toc476139394)

[Mon nouveau Hello World 12](#_Toc476139395)

[Installation de Cherrypy 12](#_Toc476139396)

[L’installation 12](#_Toc476139397)

[L’exemple de base 13](#_Toc476139398)

[Doc.cherrypy.org 14](#_Toc476139399)

[Foreword 14](#_Toc476139400)

[Why CherryPy? 14](#_Toc476139401)

# Introduction

<https://pypi.python.org/pypi/CherryPy>

<http://docs.cherrypy.org/en/latest/>

<http://sametmax.com/utiliser-cherrypy-serveur-web-leger-avec-bottle-framework-leger/>

<https://mathsp.tuxfamily.org/spip.php?rubrique110>

# Homepage

<http://cherrypy.org/>

CherryPy is as easy as…

import cherrypy

class HelloWorld(object):

def index(self):

return "Hello World!"

index.exposed = True

cherrypy.quickstart(HelloWorld())

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 very fast and stable. It is being [used in production by many sites](http://docs.cherrypy.org/en/latest/intro.html#success-stories), from the simplest to the most demanding.

Features

* A reliable, [HTTP/1.1-compliant](http://www.w3.org/Protocols/rfc2616/rfc2616-sec1.html#sec1.2), [WSGI](http://www.wsgi.org/en/latest/index.html) thread-pooled webserver.
* Easy to [run multiple HTTP servers](http://docs.cherrypy.org/en/latest/advanced.html#multiple-http-servers-support) (e.g. on multiple ports) at once.
* A [powerful configuration system](http://docs.cherrypy.org/en/latest/basics.html#config) for developers and deployers alike.
* A [flexible plugin system](http://docs.cherrypy.org/en/latest/extend.html#server-wide-functions).
* Built-in tools for [caching](http://docs.cherrypy.org/en/latest/pkg/cherrypy.lib.html#module-cherrypy.lib.caching), encoding, [sessions](http://docs.cherrypy.org/en/latest/basics.html#using-sessions), [authentication](http://docs.cherrypy.org/en/latest/basics.html#authentication), [static content](http://docs.cherrypy.org/en/latest/basics.html#static-content-serving), and [many more](http://docs.cherrypy.org/en/latest/extend.html#per-request-functions).
* Swappable and customizable...everything.
* Built-in [profiling](http://docs.cherrypy.org/en/latest/pkg/cherrypy.lib.html?highlight=profiler#module-cherrypy.lib.profiler), [coverage](http://docs.cherrypy.org/en/latest/pkg/cherrypy.lib.html?highlight=cover#module-cherrypy.lib.covercp), and testing support.
* Runs on Python 2.7+, 3.1+, PyPy, Jython and [Android](http://www.defuze.org/archives/228-running-cherrypy-on-android-with-sl4a.html).

Online tests

Each time we change our codebase, a test runs so you can see the results [here](https://travis-ci.org/cherrypy/cherrypy).

# CherryPy-10.1.1-py2.py3-none-any.whl

<https://pypi.python.org/pypi/CherryPy>

Welcome to the GitHub repository of [CherryPy](http://cherrypy.org/)!

CherryPy is a pythonic, object-oriented HTTP framework.

1. It allows building web applications in much the same way one would build any other object-oriented program.
2. This design results in less and more readable code being developed faster. It’s all just properties and methods.
3. It is now more than ten years old and has proven fast and very stable.
4. It is being used in production by many sites, from the simplest to the most demanding.
5. And perhaps most importantly, it is fun to work with :-)

Here’s how easy it is to write “Hello World” in CherryPy:

import cherrypy

class HelloWorld(object):

@cherrypy.expose

def index(self):

return "Hello World!"

cherrypy.quickstart(HelloWorld())

And it continues to work that intuitively when systems grow, allowing for the Python object model to be dynamically presented as a web site and/or API.

While CherryPy is one of the easiest and most intuitive frameworks out there, the prerequisite for understanding the [CherryPy documentation](http://docs.cherrypy.org/en/latest/) is that you have a general understanding of Python and web development. Additionally:

* Tutorials are included in the repository: <https://github.com/cherrypy/cherrypy/tree/master/cherrypy/tutorial>
* A general wiki at(will be moved to github): <https://bitbucket.org/cherrypy/cherrypy/wiki/Home>
* Plugins are described at: <http://tools.cherrypy.org/>



<https://pypi.python.org/packages/04/7e/08857376cdd0302ac4b0167f7a29d089a1b2610f91f368aa83d2bd4e346a/CherryPy-10.1.1-py2.py3-none-any.whl#md5=8b0b9da04d28cf1c270453b75eeb43a2>

# Utiliser Cherrypy (serveur web léger) avec Bottle

<http://sametmax.com/utiliser-cherrypy-serveur-web-leger-avec-bottle-framework-leger/>

Pour les sites/app qu’on developpe en une journée, [Bottle](http://bottlepy.org/docs/dev/) et [Cherrypy](http://www.cherrypy.org/) sont deux larons en foire qui s’accouplent parfaitement…

**Rappel:**

**Bottle:** Framework python light, allez voir [“Créer un site avec bottle en 5 minutes”](http://sametmax.com/creer-un-site-avec-bottle-en-5-minutes-parceque-7-cest-impossible-voyons/)  
**Cherrypy:** Framework python light + serveur web

**Une bonne** [**Pip**](http://sametmax.com/votre-python-aime-les-pip/) **pour commencer:**

|  |
| --- |
| monsieur@fion:~*# pip install cherrypy* |

**Dans votre fichier start.py:**

|  |
| --- |
| run(host='0.0.0.0', port=80, server='cherrypy') |

**Note:** le 0.0.0.0 permet d’acceder à votre site depuis l’extérieur.

Pour lancer en daemon vous pouvez utiliser [supervisord](http://supervisord.org/) **ou** faire un script dans le répertoire où se trouve start.py de votre projet et y coller:

|  |
| --- |
| *#! /bin/sh*    *### Run server in daemon mode*  nohup python start.py & |

**Plusieurs serveurs web sur la même machine:**  
Si vous avez un autre serveur web qui tourne sur votre machine et qui vous empêche de lancer cherrypy sur le port 80 vous pouvez utiliser [nginx](http://nginx.org/) en [proxy](http://wiki.nginx.org/HttpProxyModule). Pour ce faire modifiez le port de cherrypy dans votre fichier start.py (voir plus haut) et réglez-le sur 7777 par exemple avec un host 127.0.0.1.  
Ouvrez un fichier de conf nginx et copiez ceci:

|  |
| --- |
| upstream monsite\_cherrypy {  server 127.0.0.1:7777;  }    server {  listen 80;  server\_name monsite.com www.monsite.com ;    location /favicon.ico {  root /home/monsite/static/img;  }    location / {  proxy\_pass http://monsite\_cherrypy;  }    } |

Relancez Nginx (service nginx restart). Et votre site est en place. C’est ce qu’on utilise pour [multiboards](http://multiboards.net/) par exemple.

C’est rapide et simple et ça peut tenir pas mal de visiteurs. ça évite d’installer des trucs lourds avec 3 tonnes de config.

# Les Maths Libres

<https://mathsp.tuxfamily.org/spip.php?rubrique110>

## Cherrypy

Dernier ajout : 29 avril 2016.

Dans cette rubrique vous trouverez le peu de choses que je maîtrise de [Cherrypy](http://www.cherrypy.org/), une bibliothèque Python permet de créer un mini serveur web et les pages qui vont avec !

Trés léger et incroyable facile pour débuter. Pour le reste, ne m’en demandez pas trop ...

## 29 avril 2016 CherryPy, pour qui, pour quoi ?

### Cherrypy, pour qui ?

* Si comme moi, vous n’y connaissez rien en php.
* Si vous maîtrisez convenablement Python.
* Si vous souhaitez héberger un petit site chez vous.

### Cherrypy, pour quoi faire ?

* Faire tourner un site web en local en quelques minutes.
* Le rendre accessible depuis internet si vous voulez.
* Créer des pages web dynamiques en utilisant Python.
* Créer une interface web à vos projets Python.

Ça semble assez incroyable mais c’est pourtant possible. Et le principal intérêt de Cherrypy est d’être accessible aux débutants.

## 28 avril 2016 uploader un fichier

### L’idée

Cet exemple consiste à montrer comment uploader un fichier vers le serveur.

Dans mon cas,

1. on choisit le fichier ;
2. on appuie sur "valider ;
3. les informations sur le fichier s’affiche en bas de cette page ;
4. un lien permet de visualiser le fichier ;
5. un autre lien permet de le télécharger.

En pratique, le programme commence par lire le fichier sélectionné. Ensuite, il le copie dans le serveur dans le dossier "static/upload".

### Le code

import cherrypy, os

mapageIndex="""

<html>

<head>

<meta http-equiv="content-type" content="text/html; charset=utf-8" />

<title>Illustration d'un upload</title>

</head>

<body>

<h2>

Uploader un fichier

</h2>

<hr>

<form action="index" method="post" enctype="multipart/form-data">

<br>

<input type="file" name="myFile"><br>

<br>

<input type="submit" name=boutonChoisir>

</form>

<hr>

"""

mapageUpload="""

<body>

<h2>

Informations sur le fichier

</h2>

Taille: %s<br>

Nom: %s<br>

Mime-type: %s

<br>

<br>

<a href=%s target="\_blank">Visualiser le fichier</a>

<br>

<a href=%s download>T&eacute;l&eacute;charger le fichier</a>

"""

finpage="""

</body>

</html>

"""

class monSite():

#----------------- index

@cherrypy.expose

def index(self,boutonChoisir=None,myFile=None):

if boutonChoisir :

content = myFile.file.read()

nomFichier="./static/upload/" + myFile.filename

file = open (nomFichier,"wb")

file.write(content)

file.close()

return mapageIndex +mapageUpload % (len(content), myFile.filename, myFile.content\_type,nomFichier,nomFichier) + finpage

else :

return mapageIndex + finpage

cherrypy.quickstart(monSite())

## 28 avril 2016 somme de deux entiers

### Une projet, deux solutions

Je souhaite réaliser un site qui calcule la somme de deux entiers !!!!!!!! Sans commentaire SVP.

Dans la première solution, la page initiale contiendra un formulaire.  
Son action associée créera une autre page pour le résultat.

Dans la deuxième solution, le résultat apparaîtra sur la page initiale.

### Première solution

import cherrypy, os

class monSite:

#---------------index

@cherrypy.expose

def index(self) :

output="""

<form action="somme" method="GET">

<input type=number name="a" value=0> +

<input type=number name="b" value=0>

<input type="submit" value="=">

</form>

"""

return output

#-------------- somme

@cherrypy.expose

def somme(self,a,b):

output=str(int(a)+int(b))

return output

cherrypy.quickstart(monSite())

### Deuxième solution

import cherrypy, os

#---------------------------------

mapage="""

<h1>Somme de deux nombres</h1>

<p>Choisir vos deux entiers et cliquer sur le bouton.</p>

<form action="index" method="GET">

<input type=number name="a" value= %s> +

<input type=number name="b" value= %s>

<input type=submit name=bouton value="="> %s

</form>

<br>

<hr>

"""

class monSite:

#---------------index

@cherrypy.expose

def index(self,a=None,b=None,bouton=None):

if bouton:

# on calcule la somme

r = str(int(a)+int(b))

output = mapage%(a,b,r)

else:

# questionnaire initiale

output = mapage%("0","0","0")

return output

cherrypy.quickstart(monSite())

## 27 avril 2016 Un début de configuration pour Cherrypy

### Une arborescence

L’exemple de base tient en un fichier. mais pour un site plus complet, d’autres fichiers vont venir se rajouter.  
Pour cela, une arborescence est utile pour s’y retrouver.

Supposons que mon dossier principal soit "/home/dlefur/Cherrypy/".  
Dedans se trouve mon fichier **main.py**, mon fichier python à exécuter.

On peut au moins commencer par trois sous-dossiers :

* **config/** : contiendra le fichier de configuration **server.conf**.
* **favicon/** : contiendra **favicon.ico**.
* **public/** :contiendra d’autres sous-dossiers :
* **css/**
* **images/**
* **pdf/**

### Le fichier ****server.conf****

[global]

server.socket\_host : "0.0.0.0" # pour rendre le site accessible à la fois en local ou en public

server.socket\_port : 2016 # le port choisi

[/]

tools.sessions.on : True

tools.staticdir.root: "/home/dlefur/Cherrypy/"

[/favicon.ico]

tools.staticfile.on : True

tools.staticfile.filename : "/home/dlefur/Cherrypy/favicon/favicon.ico"

[/style.css]

tools.staticfile.on : True

tools.staticfile.filename : "/home/dlefur/Cherrypy/static/css/style.css"

[/static]

tools.staticdir.on : True

tools.staticdir.dir : "static/"

Le css pourra être apelé par une commande :

<link rel="stylesheet" type="text/css" href="/style.css" type="text/css"></link>

### Mon nouveau Hello World

import cherrypy, os

#---------------- le site

class HelloWorld :

#--------------------- la page par defaut

@cherrypy.expose # pour rendre la page disponible

def index(self):

return "Hello world!"

#----------------- lance le site en utilisant le fichier de configuration

configfile=os.path.join(os.path.dirname(\_\_file\_\_),"config/server.conf")

cherrypy.quickstart(HelloWorld(),config=configfile)

Cette fois-ci, le port a changé. C’est celui indiqué dans le fichier server.conf.  
Dans le navigateur, l’adresse à utiliser sera : <http://localhost:2016/>.

## [Installation de Cherrypy](https://mathsp.tuxfamily.org/spip.php?article337)

### L’installation

**Cherrypy** est une bibliothèque Python. Son installation peut se faire par le paquet **pip** qu’il faut donc commencer par installer.



On peut alors installer Cherrypy :



### L’exemple de base

Il suffit de créer un fichier **hello.py** :

import cherrypy

#---------------- le site

class HelloWorld :

#--------------------- la page par defaut

@cherrypy.expose # pour rendre la page disponible

def index(self):

return "Hello world!"

#----------------- lance le site

cherrypy.quickstart(HelloWorld())

Dans un terminal, se placer dans le dossier contenant le fichier **hello.py** et lancer la commande :

python hello.py

Dans un navigateur, rentrer l’adresse : <http://localhost:8080/>.  
Et puis c’est tout !

# Doc.cherrypy.org

<http://docs.cherrypy.org/en/latest/intro.html>

## 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.

1. 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](http://docs.cherrypy.org/en/latest/extend.html#tools) and [plugins](http://docs.cherrypy.org/en/latest/extend.html#busplugins), which are powerful extension points needed to develop world-class web applications.

1. 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.

1. 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.

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

1. It’s free!

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

1. Where to go from here?

Check out the [tutorials](http://docs.cherrypy.org/en/latest/tutorials.html#tutorials) to start enjoying the fun!

## [Installation](http://docs.cherrypy.org/en/latest/install.html#id1)

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](http://python.org/), [IronPython](http://ironpython.net/), [Jython](http://www.jython.org/) and [PyPy](http://pypy.org/)

Contents

* [Installation](http://docs.cherrypy.org/en/latest/install.html#installation)
  + [Requirements](http://docs.cherrypy.org/en/latest/install.html#requirements)
  + [Supported python version](http://docs.cherrypy.org/en/latest/install.html#supported-python-version)
  + [Installing](http://docs.cherrypy.org/en/latest/install.html#installing)
    - [Test your installation](http://docs.cherrypy.org/en/latest/install.html#test-your-installation)
  + [Run it](http://docs.cherrypy.org/en/latest/install.html#run-it)
    - [cherryd](http://docs.cherrypy.org/en/latest/install.html#cherryd)
      * [Command-Line Options](http://docs.cherrypy.org/en/latest/install.html#command-line-options)

### [Requirements](http://docs.cherrypy.org/en/latest/install.html#id2)

CherryPy does not have any mandatory requirements. However certain features it comes with will require you install certain packages. To simplify installing additional dependencies CherryPy enables you to specify extras in your requirements (e.g. cherrypy[json,routes\_dispatcher,ssl]): - doc – for documentation related stuff - json – for custom [JSON processing library](https://github.com/simplejson/simplejson) - routes\_dispatcher – [routes](http://routes.readthedocs.org/en/latest/) for declarative URL mapping dispatcher - ssl – for [OpenSSL bindings](https://github.com/pyca/pyopenssl), useful in Python environments not having the builtin [ssl](https://docs.python.org/3/library/ssl.html#module-ssl) module - testing - memcached\_session – enables [memcached](https://github.com/linsomniac/python-memcached) backend session - xcgi

### [Supported python version](http://docs.cherrypy.org/en/latest/install.html#id3)

CherryPy supports Python 2.7 through to 3.5.

### [Installing](http://docs.cherrypy.org/en/latest/install.html#id4)

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

$ easy\_install cherrypy

$ pip install cherrypy

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

$ git clone https://github.com/cherrypy/cherrypy

$ cd cherrypy

$ python setup.py install

#### [Test your installation](http://docs.cherrypy.org/en/latest/install.html#id5)

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](http://docs.cherrypy.org/en/latest/install.html#id6)

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](http://docs.cherrypy.org/en/latest/install.html#id7)

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](http://docs.cherrypy.org/en/latest/install.html#id8)

-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](http://docs.cherrypy.org/en/latest/pkg/cherrypy.process.html#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

# Doc.cherrypy.org : Tutorials

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

Contents

* [Tutorials](http://docs.cherrypy.org/en/latest/tutorials.html#tutorials)
  + [Tutorial 1: A basic web application](http://docs.cherrypy.org/en/latest/tutorials.html#tutorial-1-a-basic-web-application)
  + [Tutorial 2: Different URLs lead to different functions](http://docs.cherrypy.org/en/latest/tutorials.html#tutorial-2-different-urls-lead-to-different-functions)
  + [Tutorial 3: My URLs have parameters](http://docs.cherrypy.org/en/latest/tutorials.html#tutorial-3-my-urls-have-parameters)
  + [Tutorial 4: Submit this form](http://docs.cherrypy.org/en/latest/tutorials.html#tutorial-4-submit-this-form)
  + [Tutorial 5: Track my end-user’s activity](http://docs.cherrypy.org/en/latest/tutorials.html#tutorial-5-track-my-end-user-s-activity)
  + [Tutorial 6: What about my javascripts, CSS and images?](http://docs.cherrypy.org/en/latest/tutorials.html#tutorial-6-what-about-my-javascripts-css-and-images)
  + [Tutorial 7: Give us a REST](http://docs.cherrypy.org/en/latest/tutorials.html#tutorial-7-give-us-a-rest)
  + [Tutorial 8: Make it smoother with Ajax](http://docs.cherrypy.org/en/latest/tutorials.html#tutorial-8-make-it-smoother-with-ajax)
  + [Tutorial 9: Data is all my life](http://docs.cherrypy.org/en/latest/tutorials.html#tutorial-9-data-is-all-my-life)
  + [Tutorial 10: Make it a modern single-page application with React.js](http://docs.cherrypy.org/en/latest/tutorials.html#tutorial-10-make-it-a-modern-single-page-application-with-react-js)
  + [Tutorial 11: Organize my code](http://docs.cherrypy.org/en/latest/tutorials.html#tutorial-11-organize-my-code)
    - [Dispatchers](http://docs.cherrypy.org/en/latest/tutorials.html#dispatchers)
    - [Tools](http://docs.cherrypy.org/en/latest/tutorials.html#tools)
    - [Plugins](http://docs.cherrypy.org/en/latest/tutorials.html#plugins)

## [Tutorial 1: A basic web application](http://docs.cherrypy.org/en/latest/tutorials.html#id3)

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/>

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [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](https://docs.python.org/3/library/signal.html#module-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](http://docs.cherrypy.org/en/latest/basics.html#perappconf) to understand how to set the configuration.

## [Tutorial 2: Different URLs lead to different functions](http://docs.cherrypy.org/en/latest/tutorials.html#id4)

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | 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**](https://tools.ietf.org/html/rfc2616.html)).
* 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](http://docs.cherrypy.org/en/latest/glossary.html#term-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](http://docs.cherrypy.org/en/latest/tutorials.html#id5)

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | 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/generate> still.

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

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

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

## [Tutorial 4: Submit this form](http://docs.cherrypy.org/en/latest/tutorials.html#id6)

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | 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](http://docs.cherrypy.org/en/latest/tutorials.html#tut03) 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](http://docs.cherrypy.org/en/latest/tutorials.html#id7)

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](http://en.wikipedia.org/wiki/Session_%28computer_science%29#HTTP_session_token) that is carried during the conversation between the user and your application.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | import random  import string  import cherrypy  class StringGenerator(object):  @cherrypy.expose  def index(self):  return """<html>  <head></head>  <body>  <form method="get" action="generate">  <input type="text" value="8" name="length" />  <button type="submit">Give it now!</button>  </form>  </body>  </html>"""  @cherrypy.expose  def generate(self, length=8):  some\_string = ''.join(random.sample(string.hexdigits, int(length)))  cherrypy.session['mystring'] = some\_string  return some\_string  @cherrypy.expose  def display(self):  return cherrypy.session['mystring']  if \_\_name\_\_ == '\_\_main\_\_':  conf = {  '/': {  'tools.sessions.on': True  }  }  cherrypy.quickstart(StringGenerator(), '/', conf) |

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

$ python tut05.py

In this example, we generate the string as in the [previous](http://docs.cherrypy.org/en/latest/tutorials.html#tut04) tutorial but also store it in the current session. If you go to <http://localhost:8080/>, generate a random string, then go to <http://localhost:8080/display>, you will see the string you just generated.

The lines 30-34 show you how to enable the session support in your CherryPy application. By default, CherryPy will save sessions in the process’s memory. It supports more persistent [backends](http://docs.cherrypy.org/en/latest/basics.html#basicsession) as well.

## [Tutorial 6: What about my javascripts, CSS and images?](http://docs.cherrypy.org/en/latest/tutorials.html#id8)

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

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

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

|  |  |
| --- | --- |
| 1  2  3 | 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | 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'  }  }  cherrypy.quickstart(StringGenerator(), '/', conf) |

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

$ python tut06.py

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

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

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

## [Tutorial 7: Give us a REST](http://docs.cherrypy.org/en/latest/tutorials.html#id9)

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](http://www.ibm.com/developerworks/library/ws-restful/index.html).

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

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

Note

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | import random  import string  import cherrypy  @cherrypy.expose  class StringGeneratorWebService(object):  @cherrypy.tools.accept(media='text/plain')  def GET(self):  return cherrypy.session['mystring']  def POST(self, length=8):  some\_string = ''.join(random.sample(string.hexdigits, int(length)))  cherrypy.session['mystring'] = some\_string  return some\_string  def PUT(self, another\_string):  cherrypy.session['mystring'] = another\_string  def DELETE(self):  cherrypy.session.pop('mystring', None)  if \_\_name\_\_ == '\_\_main\_\_':  conf = {  '/': {  'request.dispatch': cherrypy.dispatch.MethodDispatcher(),  'tools.sessions.on': True,  'tools.response\_headers.on': True,  'tools.response\_headers.headers': [('Content-Type', 'text/plain')],  }  }  cherrypy.quickstart(StringGeneratorWebService(), '/', conf) |

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

$ python tut07.py

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

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

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

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

Please install [requests](http://www.python-requests.org/en/latest/) through the following command:

$ pip install requests

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | >>> import requests  >>> s = requests.Session()  >>> r = s.get('http://127.0.0.1:8080/')  >>> r.status\_code  500  >>> r = s.post('http://127.0.0.1:8080/')  >>> r.status\_code, r.text  (200, u'04A92138')  >>> r = s.get('http://127.0.0.1:8080/')  >>> r.status\_code, r.text  (200, u'04A92138')  >>> r = s.get('http://127.0.0.1:8080/', headers={'Accept': 'application/json'})  >>> r.status\_code  406  >>> r = s.put('http://127.0.0.1:8080/', params={'another\_string': 'hello'})  >>> r = s.get('http://127.0.0.1:8080/')  >>> r.status\_code, r.text  (200, u'hello')  >>> r = s.delete('http://127.0.0.1:8080/')  >>> r = s.get('http://127.0.0.1:8080/')  >>> r.status\_code  500 |

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

Lines 12-14 show you how the application reacted when our client requested the generated string as a JSON format. Since we configured the web API to only support plain text, it returns the appropriate [HTTP error code](http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.7).

Note

We use the [Session](http://www.python-requests.org/en/latest/user/advanced/#session-objects) interface of requests so that it takes care of carrying the session id stored in the request cookie in each subsequent request. That is handy.

Important

It’s all about RESTful URLs these days, isn’t it?

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

This can be easily workaround with two handy CherryPy features explained in the [advanced section](http://docs.cherrypy.org/en/latest/advanced.html#restful).

## [Tutorial 8: Make it smoother with Ajax](http://docs.cherrypy.org/en/latest/tutorials.html#id10)

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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | <!DOCTYPE html>  <html>  <head>  <link href="/static/css/style.css" rel="stylesheet">  <script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>  <script type="text/javascript">  $(document).ready(function() {  $("#generate-string").click(function(e) {  $.post("/generator", {"length": $("input[name='length']").val()})  .done(function(string) {  $("#the-string").show();  $("#the-string input").val(string);  });  e.preventDefault();  });  $("#replace-string").click(function(e) {  $.ajax({  type: "PUT",  url: "/generator",  data: {"another\_string": $("#the-string input").val()}  })  .done(function() {  alert("Replaced!");  });  e.preventDefault();  });  $("#delete-string").click(function(e) {  $.ajax({  type: "DELETE",  url: "/generator"  })  .done(function() {  $("#the-string").hide();  });  e.preventDefault();  });  });  </script>  </head>  <body>  <input type="text" value="8" name="length"/>  <button id="generate-string">Give it now!</button>  <div id="the-string">  <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](http://jquery.com/) 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | import os, os.path  import random  import string  import cherrypy  class StringGenerator(object):  @cherrypy.expose  def index(self):  return open('index.html')  @cherrypy.expose  class StringGeneratorWebService(object):  @cherrypy.tools.accept(media='text/plain')  def GET(self):  return cherrypy.session['mystring']  def POST(self, length=8):  some\_string = ''.join(random.sample(string.hexdigits, int(length)))  cherrypy.session['mystring'] = some\_string  return some\_string  def PUT(self, another\_string):  cherrypy.session['mystring'] = another\_string  def DELETE(self):  cherrypy.session.pop('mystring', None)  if \_\_name\_\_ == '\_\_main\_\_':  conf = {  '/': {  'tools.sessions.on': True,  'tools.staticdir.root': os.path.abspath(os.getcwd())  },  '/generator': {  'request.dispatch': cherrypy.dispatch.MethodDispatcher(),  'tools.response\_headers.on': True,  'tools.response\_headers.headers': [('Content-Type', 'text/plain')],  },  '/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](http://docs.cherrypy.org/en/latest/tutorials.html#id11)

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

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

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

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

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

Our application will replace the storage of the generated string from the session to a SQLite database. The application will have the same HTML code as [tutorial 08](http://docs.cherrypy.org/en/latest/tutorials.html#tut08). So let’s simply focus on the application code itself:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90 | import os, os.path  import random  import sqlite3  import string  import time  import cherrypy  DB\_STRING = "my.db"  class StringGenerator(object):  @cherrypy.expose  def index(self):  return open('index.html')  @cherrypy.expose  class StringGeneratorWebService(object):  @cherrypy.tools.accept(media='text/plain')  def GET(self):  with sqlite3.connect(DB\_STRING) as c:  cherrypy.session['ts'] = time.time()  r = c.execute("SELECT value FROM user\_string WHERE session\_id=?",  [cherrypy.session.id])  return r.fetchone()  def POST(self, length=8):  some\_string = ''.join(random.sample(string.hexdigits, int(length)))  with sqlite3.connect(DB\_STRING) as c:  cherrypy.session['ts'] = time.time()  c.execute("INSERT INTO user\_string VALUES (?, ?)",  [cherrypy.session.id, some\_string])  return some\_string  def PUT(self, another\_string):  with sqlite3.connect(DB\_STRING) as c:  cherrypy.session['ts'] = time.time()  c.execute("UPDATE user\_string SET value=? WHERE session\_id=?",  [another\_string, cherrypy.session.id])  def DELETE(self):  cherrypy.session.pop('ts', None)  with sqlite3.connect(DB\_STRING) as c:  c.execute("DELETE FROM user\_string WHERE session\_id=?",  [cherrypy.session.id])  def setup\_database():  """  Create the `user\_string` table in the database  on server startup  """  with sqlite3.connect(DB\_STRING) as con:  con.execute("CREATE TABLE user\_string (session\_id, value)")  def cleanup\_database():  """  Destroy the `user\_string` table from the database  on server shutdown.  """  with sqlite3.connect(DB\_STRING) as con:  con.execute("DROP TABLE user\_string")  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'  }  }  cherrypy.engine.subscribe('start', setup\_database)  cherrypy.engine.subscribe('stop', cleanup\_database)  webapp = StringGenerator()  webapp.generator = StringGeneratorWebService()  cherrypy.quickstart(webapp, '/', conf) |

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

$ python tut09.py

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

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

Important

In this example, we must still set the session to a dummy value so that the session is not [discarded](https://cherrypy.readthedocs.org/en/latest/pkg/cherrypy.lib.html?highlight=fixation#session-fixation-protection) on each request by CherryPy. Since we now use the database to store the generated string, we simply store a dummy timestamp inside the session.

Note

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

## [Tutorial 10: Make it a modern single-page application with React.js](http://docs.cherrypy.org/en/latest/tutorials.html#id12)

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

This tutorial demonstrates how to integrate with [React.js](https://facebook.github.io/react/), a Javascript library for SPA released by Facebook in 2013. Please refer to React.js documentation to learn more about it.

To demonstrate it, let’s use the code from [tutorial 09](http://docs.cherrypy.org/en/latest/tutorials.html#tut09). However, we will be replacing the HTML and Javascript code.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <!DOCTYPE html>  <html>  <head>  <link href="/static/css/style.css" rel="stylesheet">  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/react.js"></script>  <script src="http://code.jquery.com/jquery-2.1.1.min.js"></script>  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>  </head>  <body>  <div id="generator"></div>  <script type="text/babel" src="static/js/gen.js"></script>  </body>  </html> |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156 | var StringGeneratorBox = React.createClass({  handleGenerate: function() {  var length = this.state.length;  this.setState(function() {  $.ajax({  url: this.props.url,  dataType: 'text',  type: 'POST',  data: {  "length": length  },  success: function(data) {  this.setState({  length: length,  string: data,  mode: "edit"  });  }.bind(this),  error: function(xhr, status, err) {  console.error(this.props.url,  status, err.toString()  );  }.bind(this)  });  });  },  handleEdit: function() {  var new\_string = this.state.string;  this.setState(function() {  $.ajax({  url: this.props.url,  type: 'PUT',  data: {  "another\_string": new\_string  },  success: function() {  this.setState({  length: new\_string.length,  string: new\_string,  mode: "edit"  });  }.bind(this),  error: function(xhr, status, err) {  console.error(this.props.url,  status, err.toString()  );  }.bind(this)  });  });  },  handleDelete: function() {  this.setState(function() {  $.ajax({  url: this.props.url,  type: 'DELETE',  success: function() {  this.setState({  length: "8",  string: "",  mode: "create"  });  }.bind(this),  error: function(xhr, status, err) {  console.error(this.props.url,  status, err.toString()  );  }.bind(this)  });  });  },  handleLengthChange: function(length) {  this.setState({  length: length,  string: "",  mode: "create"  });  },  handleStringChange: function(new\_string) {  this.setState({  length: new\_string.length,  string: new\_string,  mode: "edit"  });  },  getInitialState: function() {  return {  length: "8",  string: "",  mode: "create"  };  },  render: function() {  return (  <div className="stringGenBox">  <StringGeneratorForm onCreateString={this.handleGenerate}  onReplaceString={this.handleEdit}  onDeleteString={this.handleDelete}  onLengthChange={this.handleLengthChange}  onStringChange={this.handleStringChange}  mode={this.state.mode}  length={this.state.length}  string={this.state.string}/>  </div>  );  }  });  var StringGeneratorForm = React.createClass({  handleCreate: function(e) {  e.preventDefault();  this.props.onCreateString();  },  handleReplace: function(e) {  e.preventDefault();  this.props.onReplaceString();  },  handleDelete: function(e) {  e.preventDefault();  this.props.onDeleteString();  },  handleLengthChange: function(e) {  e.preventDefault();  var length = React.findDOMNode(this.refs.length).value.trim();  this.props.onLengthChange(length);  },  handleStringChange: function(e) {  e.preventDefault();  var string = React.findDOMNode(this.refs.string).value.trim();  this.props.onStringChange(string);  },  render: function() {  if (this.props.mode == "create") {  return (  <div>  <input type="text" ref="length" defaultValue="8" value={this.props.length} onChange={this.handleLengthChange} />  <button onClick={this.handleCreate}>Give it now!</button>  </div>  );  } else if (this.props.mode == "edit") {  return (  <div>  <input type="text" ref="string" value={this.props.string} onChange={this.handleStringChange} />  <button onClick={this.handleReplace}>Replace</button>  <button onClick={this.handleDelete}>Delete it</button>  </div>  );  }  return null;  }  });  React.render(  <StringGeneratorBox url="/generator" />,  document.getElementById('generator')  ); |

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

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

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

There is not much to say and, hopefully, the meaning of that code is rather clear. The component has an internal [state](https://facebook.github.io/react/docs/interactivity-and-dynamic-uis.html) in which we store the current string as generated/modified by the user.

When the user [changes the content of the input boxes](https://facebook.github.io/react/docs/forms.html), the state is updated on the client side. Then, when a button is clicked, that state is sent out to the backend server using the API endpoint and the appropriate action takes places. Then, the state is updated and so is the view.

## [Tutorial 11: Organize my code](http://docs.cherrypy.org/en/latest/tutorials.html#id13)

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](http://docs.cherrypy.org/en/latest/extend.html#dispatchers)
* [tools](http://docs.cherrypy.org/en/latest/extend.html#tools)
* [plugins](http://docs.cherrypy.org/en/latest/extend.html#busplugins)

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

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

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

### [Dispatchers](http://docs.cherrypy.org/en/latest/tutorials.html#id14)

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](http://docs.cherrypy.org/en/latest/extend.html#dispatchers). 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](http://docs.cherrypy.org/en/latest/tutorials.html#id15)

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](http://docs.cherrypy.org/en/latest/extend.html#tools).

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](http://docs.cherrypy.org/en/latest/tutorials.html#id16)

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](http://docs.cherrypy.org/en/latest/extend.html#busplugins) are called that way because they work along with the CherryPy [engine](http://docs.cherrypy.org/en/latest/extend.html#cpengine) and extend it with your operations.

# Doc.cherrypy.org : Basics

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

Contents

* [Basics](http://docs.cherrypy.org/en/latest/basics.html#basics)
  + [The one-minute application example](http://docs.cherrypy.org/en/latest/basics.html#the-one-minute-application-example)
  + [Hosting one or more applications](http://docs.cherrypy.org/en/latest/basics.html#hosting-one-or-more-applications)
    - [Single application](http://docs.cherrypy.org/en/latest/basics.html#single-application)
    - [Multiple applications](http://docs.cherrypy.org/en/latest/basics.html#multiple-applications)
  + [Logging](http://docs.cherrypy.org/en/latest/basics.html#logging)
    - [Disable logging](http://docs.cherrypy.org/en/latest/basics.html#disable-logging)
    - [Play along with your other loggers](http://docs.cherrypy.org/en/latest/basics.html#play-along-with-your-other-loggers)
  + [Configuring](http://docs.cherrypy.org/en/latest/basics.html#configuring)
    - [Global server configuration](http://docs.cherrypy.org/en/latest/basics.html#global-server-configuration)
    - [Per-application configuration](http://docs.cherrypy.org/en/latest/basics.html#per-application-configuration)
    - [Additional application settings](http://docs.cherrypy.org/en/latest/basics.html#additional-application-settings)
  + [Cookies](http://docs.cherrypy.org/en/latest/basics.html#cookies)
  + [Using sessions](http://docs.cherrypy.org/en/latest/basics.html#using-sessions)
    - [Filesystem backend](http://docs.cherrypy.org/en/latest/basics.html#filesystem-backend)
    - [Memcached backend](http://docs.cherrypy.org/en/latest/basics.html#memcached-backend)
    - [Other backends](http://docs.cherrypy.org/en/latest/basics.html#other-backends)
  + [Static content serving](http://docs.cherrypy.org/en/latest/basics.html#static-content-serving)
    - [Serving a single file](http://docs.cherrypy.org/en/latest/basics.html#serving-a-single-file)
    - [Serving a whole directory](http://docs.cherrypy.org/en/latest/basics.html#serving-a-whole-directory)
    - [Specifying an index file](http://docs.cherrypy.org/en/latest/basics.html#specifying-an-index-file)
    - [Allow files downloading](http://docs.cherrypy.org/en/latest/basics.html#allow-files-downloading)
  + [Dealing with JSON](http://docs.cherrypy.org/en/latest/basics.html#dealing-with-json)
    - [Decoding request](http://docs.cherrypy.org/en/latest/basics.html#decoding-request)
    - [Encoding response](http://docs.cherrypy.org/en/latest/basics.html#encoding-response)
  + [Authentication](http://docs.cherrypy.org/en/latest/basics.html#authentication)
    - [Basic](http://docs.cherrypy.org/en/latest/basics.html#basic)
    - [Digest](http://docs.cherrypy.org/en/latest/basics.html#digest)
  + [Favicon](http://docs.cherrypy.org/en/latest/basics.html#favicon)

## [The one-minute application example](http://docs.cherrypy.org/en/latest/basics.html#id5)

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | import cherrypy  class Root(object):  @cherrypy.expose  def index(self):  return "Hello World!"  if \_\_name\_\_ == '\_\_main\_\_':  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](http://docs.cherrypy.org/en/latest/glossary.html#term-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](http://docs.cherrypy.org/en/latest/glossary.html#term-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](http://docs.cherrypy.org/en/latest/basics.html#id6)

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](http://docs.cherrypy.org/en/latest/basics.html#id7)

The most straightforward way is to use [cherrypy.quickstart()](http://docs.cherrypy.org/en/latest/pkg/cherrypy.html#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](http://docs.cherrypy.org/en/latest/basics.html#id8)

The [cherrypy.quickstart()](http://docs.cherrypy.org/en/latest/pkg/cherrypy.html#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()](http://docs.cherrypy.org/en/latest/pkg/cherrypy.html#cherrypy.quickstart): an [application](http://docs.cherrypy.org/en/latest/glossary.html#term-application), a hosting path segment and a configuration. The last two lines are simply starting application server.

Important

[cherrypy.quickstart()](http://docs.cherrypy.org/en/latest/pkg/cherrypy.html#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](http://docs.cherrypy.org/en/latest/advanced.html#hostwsgiapp).

## [Logging](http://docs.cherrypy.org/en/latest/basics.html#id9)

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](http://en.wikipedia.org/wiki/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](http://docs.cherrypy.org/en/latest/basics.html#id10)

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](http://docs.cherrypy.org/en/latest/basics.html#globalsettings).

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

cherrypy.config.update({'log.screen': False,

'log.access\_file': '',

'log.error\_file': ''})

### [Play along with your other loggers](http://docs.cherrypy.org/en/latest/basics.html#id11)

Your application may obviously already use the [logging](https://docs.python.org/3/library/logging.html#module-logging) module to trace application level messages. Below is a simple example on setting it up.

import logging

import logging.config

import cherrypy

logger = logging.getLogger()

db\_logger = logging.getLogger('db')

LOG\_CONF = {

'version': 1,

'formatters': {

'void': {

'format': ''

},

'standard': {

'format': '%(asctime)s [%(levelname)s] %(name)s: %(message)s'

},

},

'handlers': {

'default': {

'level':'INFO',

'class':'logging.StreamHandler',

'formatter': 'standard',

'stream': 'ext://sys.stdout'

},

'cherrypy\_console': {

'level':'INFO',

'class':'logging.StreamHandler',

'formatter': 'void',

'stream': 'ext://sys.stdout'

},

'cherrypy\_access': {

'level':'INFO',

'class': 'logging.handlers.RotatingFileHandler',

'formatter': 'void',

'filename': 'access.log',

'maxBytes': 10485760,

'backupCount': 20,

'encoding': 'utf8'

},

'cherrypy\_error': {

'level':'INFO',

'class': 'logging.handlers.RotatingFileHandler',

'formatter': 'void',

'filename': 'errors.log',

'maxBytes': 10485760,

'backupCount': 20,

'encoding': 'utf8'

},

},

'loggers': {

'': {

'handlers': ['default'],

'level': 'INFO'

},

'db': {

'handlers': ['default'],

'level': 'INFO' ,

'propagate': False

},

'cherrypy.access': {

'handlers': ['cherrypy\_access'],

'level': 'INFO',

'propagate': False

},

'cherrypy.error': {

'handlers': ['cherrypy\_console', 'cherrypy\_error'],

'level': 'INFO',

'propagate': False

},

}

}

class Root(object):

@cherrypy.expose

def index(self):

logger.info("boom")

db\_logger.info("bam")

cherrypy.log("bang")

return "hello world"

if \_\_name\_\_ == '\_\_main\_\_':

cherrypy.config.update({'log.screen': False,

'log.access\_file': '',

'log.error\_file': ''})

cherrypy.engine.unsubscribe('graceful', cherrypy.log.reopen\_files)

logging.config.dictConfig(LOG\_CONF)

cherrypy.quickstart(Root())

In this snippet, we create a [configuration dictionary](https://docs.python.org/2/library/logging.config.html#logging.config.dictConfig) that we pass on to the logging module to configure our loggers:

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

In addition, we re-configure the CherryPy loggers:

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

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

## [Configuring](http://docs.cherrypy.org/en/latest/basics.html#id12)

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](http://docs.cherrypy.org/en/latest/config.html#configindepth) around configuration.

### [Global server configuration](http://docs.cherrypy.org/en/latest/basics.html#id13)

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](http://docs.cherrypy.org/en/latest/basics.html#id14)

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

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

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

[/]

tools.gzip.on: True

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

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

class Root(object):

@cherrypy.expose

@cherrypy.tools.gzip()

def index(self):

return "hello world!"

A variant notation to the above:

class Root(object):

@cherrypy.expose

def index(self):

return "hello world!"

index.\_cp\_config = {'tools.gzip.on': True}

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

### [Additional application settings](http://docs.cherrypy.org/en/latest/basics.html#id15)

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](http://docs.cherrypy.org/en/latest/basics.html#id16)

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](http://docs.cherrypy.org/en/latest/basics.html#id17)

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](http://docs.cherrypy.org/en/latest/pkg/cherrypy.lib.html#module-cherrypy.lib.sessions) module for more details about the session interface and implementation. Notably you will learn about sessions expiration.

### [Filesystem backend](http://docs.cherrypy.org/en/latest/basics.html#id18)

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

[/]

tools.sessions.on: True

tools.sessions.storage\_class = cherrypy.lib.sessions.FileSession

tools.sessions.storage\_path = "/some/directory"

### [Memcached backend](http://docs.cherrypy.org/en/latest/basics.html#id19)

[Memcached](http://memcached.org/) is a popular key-store on top of your RAM, it is distributed and a good choice if you want to share sessions outside of the process running CherryPy.

Requires that the Python [memcached](https://pypi.org/project/memcached) package is installed, which may be indicated by installing cherrypy[memcached\_session].

[/]

tools.sessions.on: True

tools.sessions.storage\_class = cherrypy.lib.sessions.MemcachedSession

### [Other backends](http://docs.cherrypy.org/en/latest/basics.html#id20)

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

## [Static content serving](http://docs.cherrypy.org/en/latest/basics.html#id21)

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

Note

CherryPy uses the [mimetypes](https://docs.python.org/3/library/mimetypes.html#module-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](http://docs.cherrypy.org/en/latest/basics.html#id22)

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](http://docs.cherrypy.org/en/latest/basics.html#id23)

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"

### [Specifying an index file](http://docs.cherrypy.org/en/latest/basics.html#id24)

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

[/static]

tools.staticdir.on = True

tools.staticdir.dir = "/home/site/static"

tools.staticdir.index = "index.html"

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

### [Allow files downloading](http://docs.cherrypy.org/en/latest/basics.html#id25)

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](http://docs.cherrypy.org/en/latest/basics.html#id26)

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

### [Decoding request](http://docs.cherrypy.org/en/latest/basics.html#id27)

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](http://docs.cherrypy.org/en/latest/basics.html#id28)

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](http://docs.cherrypy.org/en/latest/basics.html#id29)

CherryPy provides support for two very simple authentication mechanisms, both described in [**RFC 2617**](https://tools.ietf.org/html/rfc2617.html): Basic and Digest. They are most commonly known to trigger a browser’s popup asking users their name and password.

### [Basic](http://docs.cherrypy.org/en/latest/basics.html#id30)

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

from cherrypy.lib import auth\_basic

USERS = {'jon': 'secret'}

def validate\_password(realm, username, password):

if username in USERS and USERS[username] == password:

return True

return False

conf = {

'/protected/area': {

'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](http://docs.cherrypy.org/en/latest/basics.html#id31)

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](http://docs.cherrypy.org/en/latest/basics.html#id32)

CherryPy serves its own sweet red cherrypy as the default [favicon](http://en.wikipedia.org/wiki/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](http://docs.cherrypy.org/en/latest/basics.html#staticontent) 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)