CherryPy Tutorial

Release 0.10

Remi Delon

March 19, 2004

Email: remi@cherrypy.org

Copyright © 2002-2004 CherryPy Team (team@cherrypy.org) All rights reserved.
See the end of this document for complete license and permissions information.

Abstract

CherryPy is a Python based web development toolkit. It provides all the features of an enterprise-class application server while remaining light, fast and easy to learn.

CherryPy allows developers to build web applications in much the same way they would build any other object-oriented Python program. This usually results in smaller source code developed in less time.

It runs on most platforms (everywhere Python runs) and it is available under the GPL license.

CherryPy is now nearly two years old and it is has proven very fast and stable. It is being used in production by many sites, from the simplest ones to the most demanding ones.

Oh, and most importantly: CherryPy is fun to work with :-)

CONTENTS

CHAPTER

ONE

Downloading, installing and running the demo

1.1 Prerequisite

CherryPy uses 100% pure Python, so it runs everywhere Python runs (Windows, Unix, Mac, ...). All you need is a working distribution of Python 2.1 or higher on your machine.

1.2 Downloading and installing

Download the latest version of CherryPy from this page http://www.cherrypy.org/download and unzip/untar the file

This will create a **CherryPy-version/** directory with a few files and sub-directories

1.3 Running the demo

Go to the 'demo/' directory and type the following command:

```
python ../cherrypy.py Root.cpy
```

[1] [2]

This will create a file called 'RootServer.py'

To run it, just enter:

[1] '`python`' must be in your PATH and it must be version 2.1 or higher
[2] If *make* is installed on your machine, you can just type '`make`'

```
[remi@serveur demo]$ python RootServer.py
Reading parameters from RootServer.cfg ...
Calling initServer() ...
Reading gaz station database for the Prestation demo ...
Done reading database
Starting Httpd service ...
Server parameters:
portOrUnixFile: 8000
numberOfProcs: 1
staticContentList: [('images', 'images')]
Serving on 8000
```

Then open a web browser an type in the URL: http://localhost:8000/, and voila

You can play around with the demo website called **Prestation** or with the little tests that demonstrate a few of CherryPy's capabilities.

1.4 Stopping the server

In most cases, you can just stop the server by hitting "Ctrl-C".

This will not work if you're using Python-2.1 or Python-2.2 on Windows and the *timeoutsocket.py* module is not installed. In that case, you can stop the server by hitting "Ctrl-Break".

If you're on a unix-based system and the server is running in the background, then you can stop it by typing "kill <PID>".

Now that we've downloaded CherryPy and run the demo, it's time to understand how a developer can develop a website using CherryPy...

CHAPTER

TWO

Concepts used in CherryPy

2.1 Creation of a website

CherryPy sits between a compiler and an application server.

- Like a compiler, it reads input files and generates an executable. The executable contains everything to run the website, including its own HTTP server.
- Like an application server, it delivers highly-dynamic web sites that can be linked to other ressources, like a database for instance.

2.2 Handling of requests

In a server generated by CherryPy, every request from a client (for instance, a browser requesting an URL) is transformed into a call to the method of a class. The parameters sent by the client become the arguments of the function.

With CherryPy, website developers just have to implement those classes and those methods. It doesn't matter if the parameters are sent with a GET, a POST, if they're a short string or a large file that's being uploaded. They're all converted to a regular Python string and passed as an argument to the method. It's all transparent to the developer.

2.3 Programming a website

Input files for CherryPy are written using an extension to the Python language. This extension defines some special classes called **CherryClass**. It also defines different types of methods for those CherryClasses:

- **functions**: they are used to process some data. They are written in regular Python. Functions typically take some data as input and return some data (as opposed to, say, HTML) as output.
- **masks**: they are used to render some data. They are written in CHTL or CGTL (which are CherryPy's templating languages). Masks typically take some data as input and return HTML (or XML; Javascript, ...) as output.
- **views**: a view is written in regular Python. Views can be used in two different ways:
  - they can be used like masks, to render some data. In this case, the only difference with masks is the language they're written in. For instance, a page with lots of static data and only a little bit of dynamic data will be best written as a mask (in CHTL or CGTL). A page with lots of dynamic data and only a bit of static data will be best written as a view (in Python).
  - they can be used as the link between a function and a mask. In this case, the source code of the view will typically be: **apply this mask to the result of that function**

This concept of functions, masks and views used in CherryClasses is one of the main feature of CherryPy. A CherryClass can contain all the information to process some data and to display the result, making it a self-contained module that can be easily reused or sub-classed.

We've seen a few of the powerful concepts used in CherryPy. More concepts will be described later, but it's now time to create our first website...

CHAPTER

THREE

Creating a first website: Hello, world !

At the same level as the 'demo/' directory, create a directory called 'hello/'.

Go to the 'hello/' directory and create a file called 'Hello.cpy' that contain the following lines:

```
CherryClass Root:
mask:
    def index(self):
        <html><body>
            Hello, world !
        </body></html>
```

[1]

To compile the file, type:

```
python ../cherrypy.py Hello.cpy
```

This will create a file called 'HelloServer.py', which contains everything to run the website (including an HTTP server). To start it, just type:

```
python HelloServer.py
```

To see the page, open a browser and type in the URL: http://localhost:8000/

What we've learned:

- Source files for CherryPy are written using an extended version of the Python language (some parts use CherryPy's templating language)
- Source filenames for CherryPy have a '.cpy' extension and start with an upper case letter
- Just like any Python source file, CherryPy source files are indentation-sensitive. See the footnotes to find out more about how CherryPy handles indentation.
- The `CherryClass` keyword is used just like the `class` keyword in Python. The name of the CherryClass must start with an upper case letter.

[1] You can either use 4 whitespaces or one TAB to indent your code. It is possible to use more or less than 4 whitespaces for indenting (for instance, 3 whitespaces) by using the -W option to tell CherryPy about it (for instance: -W 3). Please note that, unlike Python, one tab can never correspond to 2 indentation levels. It always corresponds to one indentation level.

- Inside a CherryClass, you can define different sections, like `mask`, `view` or `function`. We'll see later on how they are used an what they mean.
- Inside a section, you can define methods just like you would in Python. (i.e: `def index(self):`)
- The body for a mask method isn't written in Python. Instead, it's written in CHTL or CGTL which are CherryPy's templating languages. We'll learn more about those languages later on.
- The file generated by CherryPy from an input file 'Foo.cpy' is called 'FooServer.py'
- The file generated by CherryPy is 100% pure Python
- When the browser requests the page at the root of the website, `root.index` gets called and its return value is being sent to the browser

Now let's add some dynamic functionality to it...

CHAPTER

FOUR

Creating a first dynamic website: Hello, you !

Edit the file ‘Hello.cpy’ that we’ve created in the previous chapter, and change it to:

```
CherryClass Root:
mask:
    def index(self, name="you"):
        <html><body>
            Hello, <b py-eval="name"></b> !
            <form py-attr="request.base" action="" method="get">
                Enter your name: <input name=name type=text><br>
                <input type=submit value=OK>
            </form>
        </body></html>
```

Recompile the file and restart the server. Now, refresh the page in your browser. You should see

```
Hello, you
```

followed by a field where you can enter some text. Enter your name and press the OK button. Now the string has changed to

```
Hello, "your name"
```

How does it work ?

This time, the *index* method has a parameter called *name*. Just like for any Python method, this parameter can have a default value (in this case, *you*). The first time the browser displays the page, it doesn’t pass any *name* parameter, so *name* will have its default value in the function.

When you fill out the text field and hit OK, the browser will request the same page, but this time, *name* will be passed as a parameter and it will contain the name you entered.

Because we used *method="get"* in the form, the name parameter will be passed using the URL (you can check that the URL in your browser looks like: http://localhost:8000/?name=yourName).

Now, edit ‘Hello.cpy’ and change *method="get"* to *method="post"*. Recompile the file, restart the server and redo the test: it works exactly the same way, except that *name=yourName* doesn’t show up in the URL. This is because we used a *POST* method instead of a *GET* method for the form.

What we've learned:

- Parameters sent by the browser using the URL (GET) or using a POST are passed to the method via regular Python parameters. It doesn't matter if it's a GET or a POST, or if the parameter is a short string or a large file. It will always be handled the same way
- In a mask method, you can use special tags like `py-eval` or `py-attr`. Those tags are part or the CherryPy templating languages.

It's now time to do more intersting things with the templating languages...

CHAPTER

FIVE

Templating languages: CHTL and CGTL

CherryPy comes with TWO templating languages. They are very similar to each other, but they are used in different situations. They have been designed to be very easy to use, and yet very powerful.

- **CherryPy HTML Templating Language (CHTL)**: It is used to produce HTML output. It has been designed so that HTML editors keep the dynamic informations in the page. This way, webdesigners can edit the pages using their favorite HTML editor without losing the information that the developers put in them. The trick is to hide code in the attributes of HTML tags.
- **CherryPy Generic Templating Language (CGTL)**: It is used to produce non-HTML output (for instance Javascript, CSS, XML, ...). It is very close to CHTL, except that we cannot use HTML tags to hide the code, and the syntax is a bit simpler. **If you don't use any HTML editor to create your pages, then you should probably use CGTL only as it is simpler than CHTL**.

Note that you can also use other templating languages if you want (for instance, there is a HowTo that explains how to use Cheetah), or you can just use plain python to generate your pages.

5.1 Tags

Both languages use only 6 tags: `py-eval`, `py-attr`, `py-exec`, `py-code`, `py-if` (with `py-else`) and `py-for`.

All tags are used the same way: `CherryPyTag="Some Python code"`. For instance:

```
py-eval="2*2"
py-exec="a=1"
py-if="2==0"
py-for="i in range(10)"
...
```

If you want to use double-quotes inside the python code, you need to escape them using a backslash, like this:

```
py-eval="'I love \"cherry pies\"'"
```

Let's see what each of these tags is used for:

5.1.1 py-eval

This tag is used to evaluate a Python expression, like this:

```
Hello, the sum is <py-eval="1+2+3">
```

This line will be rendered as:

```
Hello, the sum is 6
```

What happens is that CherryPy first evaluates the expression (using *eval*) and then uses *str* on the result to convert it to a string.

All the CGTL tags can be closed using /> or / >, like this:

```
<py-eval="'abcd' * 2" />
<py-eval="'Hello, %s' % name"/>
```

In its CHTL form, the *py-eval* tag can be used with any pair of opening/closing tags, like this:

```
<span class=myClass py-eval="2*3"></span>
<a href="myHref" py-eval="'Click here'"></a>
<u py-eval="'I\'m with an underline'"></u>
...
```

If you don't want to use any HTML tag around the expression, the trick is to use the `<div>` HTML tag:

```
This is a long string with a <div py-eval="'variable'"></div> in it
```

5.1.2 py-attr

This tag is like `py-eval`, except it's used for HTML tag attributes. The way it is used is as follows:

```
<td py-attr="'red'" bgColor="">
```

This will be rendered as

```
<td bgColor="red">
```

Note that this is equivalent to:

```
<td bgColor="<py-eval="'red'">">
```

But the first syntax is preferred.

5.1.3 py-exec and py-code

These tags are used to execute some Python code. `py-exec` is used to execute one line of Python code, and `py-code` is used to execute blocks of Python code. For instance, the following code:

```
<py-exec="a=2">
<py-code="
    if a==2:
        b=1
    else:
        b==2
">
b equals <py-eval="b">
```

Will be rendered as:

```
b equals 1
```

In the CHTL syntax, both tags have to be embedded in and tags as follow:

```
<div py-exec="a=2"></div>
<div py-code="
    if a==2:
        b=1
    else:
        b=2
"></div>
```

If you want to render some data inside the Python code, you must append it to the *_page* variable:

```
<html><body>
    Integers from 0 to 9:
    <py-code="
        for i in range(10):
            _page.append("%s "%i)
    ">
</body></html>
```

This will be rendered as:

```
<html><body>
    Integers from 0 to 9:
    0 1 2 3 4 5 6 7 8 9
</body></html>
```

5.1.4 py-if and py-else

These tags are used like `if` and `else` in Python. The syntax is as follows:

```
<py-if="1==1">
    OK
</py-if><py-else>
    Not OK
</py-else>
```

This will be rendered as

```
OK
```

Note that if there is an `else` clause, the `<div py-else>` tag must follow the `</div>` tag closing the `<div py-if` tag, with no significant characters in between (ie: only separators are allowed)

The CHTL equivalent is:

```
<div py-if="1==1">
    OK
</div>
<div py-else>
    Not OK
</div>
```

5.1.5 py-for

This tag is used like `for` in Python. The syntax is as follows:

```
<py-for="i in range(10)">
    <py-eval="i">
</py-for>
```

This will be rendered as

```
0 1 2 3 4 5 6 7 8 9
```

Note that you can loop over a list of tuples:

```
<py-for="i,j in [(0,0), (0,1), (1,0), (1,1)]">
    <py-eval="i+j">
</py-for>
```

This will be rendered as

```
0 1 1 2
```

The CGTL equivalent is:

```
<div py-for="i,j in [(0,0), (0,1), (1,0), (1,1)]">
    <div py-eval="i+j"></div>
</div>
```

In a *py-for* loop, CherryPy sets two handy special variables for you: *_index* and *_end*. The former is an integer containing the current number of iterations (from 0 to n-1). The latter contains the total number of iteration minus one.

For instance, if you want to display a list with the first item in bold and the last item underlined, you can use the following code:

```
<py-exec="myList=[1,5,3,2,5,4,5]">
<py-for="item in myList">
    <py-if="_index==0"><b py-eval="item"></b>
    </py-if><py-else>
        <py-if="_index==_end"><u py-eval="item"></u>
        </py-if><py-else><py-eval="item"></py-else>
        </div>
    </py-else>
</py-for>
```

This will be rendered as:

```
<b>1</b> 5 3 2 5 4 <u>5</u>
```

In the next section, we'll see how to use all these tags together...

5.2 Putting it together

We are going to make a web page that displays a table with all HTML colors. Edit the Hello.cpy file and change it to:

```
CherryClass Root:
mask:
    def index(self):
        <html><body>
            <a py-attr="request.base+'/webColors'" href="">
                Click here to see a nice table with all web colors
            </a>
        </body></html>
    def webColors(self):
        <html><body>
            <py-exec="codeList=['00', '33', '66', '99', 'CC', 'FF']">
            <table border=1>
            <py-for="r in codeList">
                <py-for="g in codeList">
                    <tr>
                        <py-for="b in codeList">
                            <py-exec="color='#%s%s%s'%(r,g,b)">
                            <td py-attr="color" bgColor="" py-eval="'  '+color+' 
                        </py-for>
                    </tr>
                </py-for>
            </py-for>
        </body></html>
```

Recompile the file, restart the server and refresh the page in your browser. Click on the link and you should see a nice table with all web colors.

How does it work ?

The `webColors` method is a pretty straight forward use of the CHTL tags. One more interesting line is:

```
<a py-attr="request.base+'/webColors'" href="">
```

request is a global variable set by CherryPy for each request of a client. It is an instance of a class with several variables. One of them is called `base` and contains the base URL of the website (in our case: http://localhost:8000). So, the line

```
<a py-attr="request.base+'/webColors'" href="">
```

will be rendered as:

```
<a href="http://localhost:8000/webColors">
```

This also tells us that when the browser requests the URL http://localhost:8000/webColors, the *webColors* method of the *Root* class gets called.

In the next chapter, we'll learn how to use views, functions and when we should use them...

CHAPTER

SIX

Views, functions and more

So far, we've only used one kind of method: masks. We are going to learn how to use views and functions.

6.1 Different architectures for a web site source code

We'll take two examples to show you two different ways to design the architecture of your code.

6.1.1 First example: straightforward architecture

Let's assume that you want to build a very simple web site where people can look for books and see the details about one perticular book. The web site is made of two kinds of pages:

- The main page that displays the list of books. Each book name is a link.
- A page that displays the informations about a perticular book. This page is displayed then the user clicks on a book name

To implement this web site, we'll just use 2 functions and 2 masks:

- One function called **getBookListData** that returns a list of book names
- One function called **getBookData** that returns the detailed informations about a perticular book
- One mask called **index** that displays the list of book names.
- One mask called **displayBook** that displays the detailed informations about a perticular book

The code of the web site is:

```
CherryClass Root:
variable:
    # Sample book list data. In real life, this would probably come from a database
    # (title, author, price)
    bookListData=[
        ('Harry Potter and the Goblet of Fire', 'J. K. Rowling', '9$'),
        ('The flying cherry', 'Remi Delon', '5$'),
        ('I love cherry pie', 'Eric Williams', '6$'),
        ('CherryPy rules', 'David stults', '7$')
    ]

function:
    def getBookListData(self):
        return self.bookListData
    def getBookData(self, id):
        return self.bookListData[id]
mask:
    def index(self):
        <html><body>
            Hi, choose a book from the list below:<br>
            <py-for="title, dummy, dummy in self.getBookListData()">
                <a py-attr="'displayBook?id=%s'%_index" href="" py-eval="title"></a><br>
            </py-for>
        </body></html>
    def displayBook(self, id):
        <html><body>
            <py-exec="title, author, price=self.getBookData(int(id))">
            Details about the book:<br>
            Title: <py-eval="title"><br>
            Author: <py-eval="author"><br>
            Price: <py-eval="price"><br>
        </body></html>
```

As you can see, the code for this "mini" web site is pretty straightforward: each mask corresponds to a page type. Since we have 2 types of pages, we use 2 masks.

Let's take a slightly more complicated example ...

6.1.2 Second example: more elegant architecture for more complex web sites

In this example, we'll add a few more features to our web site:

- This time, we want our web site to come in two languages: English and French
- In addition to being able to browse the books by title, we also want to be able to browse them by author

This now means that we have six types of pages:

- 1. View book list in English broken up by title
- 2. View book list in French broken up by title
- 3. View book list in English broken up by author
- 4. View book list in French broken up by author
- 5. View book details in English

- 6. View book details in French

If we were to keep the same architecture as the first example, we would have to write 6 masks (plus the functions). Let's try to do better than that ...

There isn't much we can do about the last 2 types of pages (5 and 6). But for the first four, we can in fact use 2 functions and 2 masks. By combining each function with each mask, we have our 4 combinations (2 times 2). We'll use the following:

- One function called **getBookListByTitleData** that returns a list of book broken up by title
- One function called **getBookListByAuthorData** that returns a list of book broken up by author
- One mask called **bookListInEnglishMask** that displays a list of books in English (it doesn't matter if the books are broken up by title or by author).
- One mask called **bookListInFrenchMask** that displays a list of books in French.

In order to "link" a mask with a function, we'll use a view. This means that we have 4 views, one for each combination. Each view will have a very simple code: **apply this mask to the result of that function**

The code for our web site looks like this:

```
CherryClass Root:
variable:
    # Sample book list data. In real life, this would probably come from a database
    # (title, author, price)
    bookListData=[
        ('Harry Potter and the Goblet of Fire', 'J. K. Rowling', '9$'),
        ('The flying cherry', 'Remi Delon', '5$'),
        ('I love cherry pie', 'Eric Williams', '6$'),
        ('CherryPy rules', 'David Stults', '7$')
    ]

function:
    def getBookListByTitleData(self):
        titleList=[]
        for title, dummy, dummy in self.bookListData: titleList.append(title)
        return titleList
    def getBookListByAuthorData(self):
        authorList=[]
        for dummy, author, dummy in self.bookListData: authorList.append(author)
        return authorList
    def getBookData(self, id):
        return self.bookListData[id]
mask:
    def bookListInEnglishMask(self, myBookListData):
            Hi, choose a book from the list below:<br>
            <py-for="data in myBookListData">
                <a py-attr="'displayBookInEnglish?id=%s'%_index" href="" py-eval="data"></a><br
            </py-for>
            <br>
    def bookListInFrenchMask(self, myBookListData):
            Bonjour, choisissez un livre de la liste:<br>
            <py-for="data in myBookListData">
                <a py-attr="'displayBookInFrench?id=%s'%_index" href="" py-eval="data"></a><br>
            </py-for>
            <br>
    def displayBookInEnglish(self, id):
        <html><body>
            <py-exec="title, author, price=self.getBookData(int(id))">
            Details about the book:<br>
            Title: <py-eval="title"><br>
            Author: <py-eval="author"><br>
            Price: <py-eval="price"><br>
            <br>
            <a py-attr="'displayBookInFrench?id=%s'%id" href="">Version francaise</a>
        </body></html>
    def displayBookInFrench(self, id):
        <html><body>
            <py-exec="title, author, price=self.getBookData(int(id))">
            Details du livre:<br>
            Titre: <py-eval="title"><br>
            Auteur: <py-eval="author"><br>
            Prix: <py-eval="price"><br>
            <br>
            <a py-attr="'displayBookInEnglish?id=%s'%id" href="">English version</a>
        </body></html>
view:
    def englishByTitle(self):
        page="<html><body>"
        byTitleData=self.getBookListByTitleData()
        page+=self.bookListInEnglishMask(byTitleData)
        page+='<a href="englishByAuthor">View books by author</a><br>'
        page+='<a href="frenchByTitle">Version francaise</a>'
        page+="</body></html>"
        return page
    def frenchByTitle(self):
```

Alternatively, we could save even more lines of code by passing the language (French or English) and the type of list (title or author) as parameters. This way, we wouldn't need to use views, and the masks could be called directly...

6.2 More examples on using functions, masks and views together

In this section, we'll build a small website thats prompts the user for an integer N between 20 and 50, and for a number of columns C between 2 and 10. Then it will display integers from 1 to N in a table of C columns.

Edit 'Hello.cpy' and enter the following code:

```
CherryClass Root:
function:
    def prepareTableData(self, N, C):
        # Prepare data that will be rendered in the table
        # Example, for N=10 and C=3, it will return:
        # [[1,2,3],
        #  [4,5,6],
        #  [7,8,9],
        #  [10]]
        N=int(N)
        C=int(C)
        tableData=[]
        i=1
        while 1:
            rowData=[]
            for c in range(C):
                rowData.append(i)
                i+=1
                if i>N: break
            tableData.append(rowData)
            if i>N: break
        return tableData

view:
    def viewResult(self, N, C):
        tableData=self.prepareTableData(N,C)
        return self.renderTableData(tableData)
mask:
    def renderTableData(self, tableData):
        # Renders tableData in a table
        <html><body>
        <table border=1>
            <div py-for="rowData in tableData">
                <tr>
                    <div py-for="columnValue in rowData">
                        <td py-eval="columnValue"></td>
                    </div>
                </tr>
            </div>
        </table>
        </body></html>

    def index(self):
        <html><body>
            <form py-attr="request.base+'/viewResult'" action="">
                Integer between 20 and 50: <input type=text name=N><br>
                Number of columns between 2 and 10: <input type=text name=C><br>
                <input type=submit>
            </form>
        </body></html>
```

How does it work ?

The *index* mask is easy to understand and is only used to input N and C.

The *prepareTableData* function is used to process N and C and to compute a list of list that will be ready to render. The *renderTableData* mask takes as an input the return value of *prepareTableData* and renders it. The *viewResult* view

is the link between the two. It basically says to compute the result of a function and to apply a mask to it.

Now, what if we want to display the integers by column instead of displaying them by line ?

Well, we just need to create a new mask, and to update the view in order to apply the new mask to the data.

Modify ‘Hello.cpy’ as follows:

```
CherryClass Root:
function:
    def prepareTableData(self, N, C):
        N=int(N)
        C=int(C)
        tableData=[]
        i=1
        while 1:
            rowData=[]
            for c in range(C):
                rowData.append(i)
                i+=1
                if i>N: break
            tableData.append(rowData)
            if i>N: break
        return tableData

view:
    def viewResult(self, N, C, displayBy):
        tableData=self.prepareTableData(N,C)
        if displayBy=="line": mask=self.renderTableDataByLine
        else: mask=self.renderTableDataByColumn
        return mask(tableData)
mask:
    def renderTableDataByLine(self, tableData):
        <html><body>
        <table border=1>
            <div py-for="rowData in tableData">
                <tr>
                    <div py-for="columnValue in rowData">
                        <td py-eval="columnValue"></td>
                    </div>
                </tr>
            </div>
        </table>
        </body></html>
    def renderTableDataByColumn(self, tableData):
        <html><body>
        <table border=1>
            <tr>
                <div py-for="rowData in tableData">
                    <td valign=top>
                        <div py-for="columnValue in rowData">
                            <div py-eval="columnValue"></div><br>
                        </div>
                    </td>
                </div>
            </tr>
        </table>
        </body></html>

    def index(self):
        <html><body>
            <form py-attr="request.base+'/viewResult'" action="">
                Integer between 20 and 50: <input type=text name=N><br>
                Number of columns (or lines) between 2 and 10: <input type=text name=C><br>
                Display result by: <select name=displayBy>
                    <option>line</option>
                    <option>column</option>
                </select><br>
                <input type=submit>
            </form>
        </body></html>
```

We've rename the *renderTableData* mask into *renderTableDataByLine*, and we've added a new mask called *renderTableDataByColumn*. *viewResult* now has a *displayBy* parameter, that's entered by the user. Based on that, *viewResult* selects which mask to apply and applies it to the result of the *prepareTableData* function (which hasn't changed).

Now, try one more test: In your browser, enter the URL: http://localhost:8000/prepareTableData?N=30&C=5

You should see the following error:

```
CherryError: CherryClass "root" doesn't have any view or mask function called "prepareTableData
```

This means that a function cannot be "called" directly from a browser. Only views and masks, which return some rendered data, can be "called" directly.

What we've learned:

- CherryPy allows true separation of content and presentation by using functions, masks and views
- Functions process some data and return some data. A function implementing an algorithm can be reused on all kinds of data.
- Masks take some data as input and render it. A mask can be reused for all kinds of data
- Views are the link between functions and masks.

Note: inside a CherryClass declaration, the different sections (function, mask or view) can appear in any order, as many times as you want.

In the next chapter, we'll see how CherryPy determines which method to call based on the URL...

CHAPTER

SEVEN

Class, instance, method and URL

So far, we've seen that the URL http://localhost:8000 triggers a call to the *index* method of the *Root* class. The URL http://localhost:8000/viewResult triggers a call to the *viewResult* method of the *Root* class.

- How does this magic work ?
- I know it cannot call the method of a class. It has to call the method of **an instance** of a class. How do you explain that ?
- What if I type the URL http://localhost:8000/dir1/dir2/dir3/page

Let's answer these questions:

First of all, when you declare a CherryClass in the source file:

```
CherryClass ClassName:
```

CherryPy (or, to be more precise: the executable generated by CherryPy) will automatically create an instance of this class, called *className*. This is the name of the class with a lower case first letter. (This is the reason why CherryClass names should always start with an upper case letter)

This instance is a global variable and can be acces from anywhere in the program.

Based on the URL, how does CherryPy know which method of which class instance to call ?

It uses a simple mechanism: For the URL *host/dir1/dir2/dir3/page*, it will call the *dir1_dir2_dir3.page()* method. So it will expect your program to have a CherryClass called *Dir1_dir2_dir3*, which should have a method called *page*.

There are 2 special cases:

- if there is no first directory (in other words, the URL is in the form: *host/page*, then it will call the *root.page()* method. This means that both URL *host/page* and *host/root/page* are perfectly equivalent.
- if there is no first directory and no page (in other words, the URL is just the host name), then it will call the *root.index()* method.

Important: In CherryPy-0.9, a new feature was added: if the page http://localhost:8000/dir1/dir2/dir3 is requested, CherryPy will convert it first to *dir1_dir2.dir3()*, so it will expect a *dir3* method in the *Dir1_dir2* CherryClass. But if no such method exists, then it will look for an *index* method in the *Dir1_dir2_dir3* CherryClass. (this would also correspond to http://localhost:8000/dir1/dir2/dir3/index).

In the next chapter, we'll see how to use inheritance when we have similar modules inside a website...

CHAPTER

EIGHT

Using OOP to program a website

One the the most powerful feature of CherryPy is that you can really use an **object oriented** approach to "program" your website.

When you look at a complex website, you'll realize that some parts have a lot in common:

- They use the same functionalities but these functionalities are applied to a different type of data
- They display the same information, but the data is displayed differently (for instance, when you have multiple versions of a website, or you have several versions in several languages)

In both cases, OOP provides an elegant solution to the problem and minimizes the amount of code that is required to implement the solution.

To show you how this can be done, we'll create a website that comes in two versions: the English version and the French version. Not only do the text and labels change, but also the colors and the way modules are displayed.

Enter the following code for the website:

```
    ########################
    CherryClass Airline abstract:
    ########################
    function:
        def localize(self, stri):
            return self.dictionnary.get(stri, stri)
    mask:
        def header(self):
            <html><body>
                <center>
                    <H1 py-eval="self.localize('Welcome to CherryPy airline')"></H1>
                    <div py-if="self==airlineFrench">
                        <a py-attr="request.base+'/airlineEnglish/index'" href="">
                            Click here for English version
                        </a>
                    </div><div py-else>
                        <a py-attr="request.base+'/airlineFrench/index'" href="">
                            Cliquez ici pour la version française
                        </a>
                    </div>
                    <br><br><br><br>
        def footer(self):
                </center>
            </body></html>
        def squareWithText(self, title, text):
            <table border=0 cellspacing=0 cellpadding=1 width=200><tr>
                <td py-attr="self.borderColor" bgColor="">
                    <table border=0 cellspacing=0 cellpadding=5><tr>
                        <td py-attr="self.insideColor" bgColor=""
                            align=center width=198 py-eval="'<b>%s</b><br><br>%s'%(title,text)">
                        </td>
                    </tr></table>
                </td>
            </tr></table>
    view:
        def bookAFlight(self):
            page=self.header()
            page+=self.squareWithText(self.localize('Booking a flight'),
                self.localize('To book a flight, think about where you want to go, and you should d
            page+=self.footer()
            return page


    ########################
    CherryClass AirlineFrench(Airline):
    ########################
    variable:
        insideColor='#FFFF99'
        borderColor='#FF6666'
        dictionnary={
            'Welcome to CherryPy airline': 'Bienvenue chez CherryPy airline',
            'Booking a flight': 'Réserver un vol',
            'To book a flight, think about where you want to go, and you should dream about it toni
                'Pour réserver un vol, pensez très fort à la destination, et vous devriez en rêver
        }
    view:
        def index(self):
            page=self.header()
            page+=self.squareWithText('Réserver un vol', 'Pour réserver un vol, cliquez sur <a href
            page+=self.squareWithText('Présentation', 'CherryPy airline est la compagnie qui vous e
            page+=self.footer()
            return page

    ########################
    CherryClass AirlineEnglish(Airline):
```

This program uses a lot of new features of CherryPy. Let's try to understand how it works:

The idea is to use a generic CherryClass (*Airline*) that contains functions, masks and views that are common to both versions (English and French) or the website. Then we use 2 CherryClasses (*AirlineFrench* and *AirlineEnglish*) to implement specificities of each version.

We use two different ways to implement specificities of each version:

- Either the only difference is the labels (they are translated into French for the French version). In this case, we just use a dictionnary to implement the translation.
- Or the presentation of the information also changes. In this case, we write 2 versions of each method.

This example also shows some new features of CherryPy:

- **Abstract CherryClasses**: in this example, you'll notice that CherryClass *Airline* is declared "abstract". This just means that the server won't create any instance of the CherryClass called *airline*. As a result, you can't use the URL http://localhost:8000/airline/index. The idea behind this is that *Airline* cannot be used directly. It has to be derived into sub-classes, and only these sub-classes can be accessed from the browser.
- **Variable section**: you'll notice that this example uses a new kind of section in a CherryClass: *variable*. This is just used to set variables by default for this CherryClass.
- **self.getPath()**: by default, CherryPy creates a method called *getPath* for each CherryClass. This method returns the URL of the CherryClass. For instance, for the *AirlineFrench* CherryClass, *getPath* would return http://localhost:8000/airlineFrench.

  This method also takes an optional argument *includeRequestBase*, which defaults to 1. If you set it to 0 then the returned URL won't include the domain name. For instance, for the *AirlineFrench* CherryClass, *getPath(0)* would return /airlineFrench.

In the next chapter, we'll learn how to split our code in several source files...

CHAPTER

NINE

Using several modules

As you'll program bigger websites, you'll soon feel the need to split your source code in several modules. There are two ways to do this:

- Either your modules are completely independant. In this case, just create your files (for instance: 'Hello1.cpy', 'Hello2.cpy' and 'Hello3.cpy') and compile them using:

  ```
  python ../cherrypy.py Hello1.cpy Hello2.cpy Hello3.cpy
  ```

  Note that the executable that CherryPy will generate will be called 'Hello1Server.py'

- Or one module is needed by the other (for instance, one is a library used by the other one). In this case, all you need to do is type the keyword *use module* on the very first line of the file. This works like an *import* statement in Python. For instance, you can have:

  ```
  ***** File BoldTime.cpy: *****
  import time

  CherryClass BoldTime:
  view:
      def getBoldTime(self):
          # Display the time in bold
          return "<b>%s</b>"%time.time()

  ***** File Hello.cpy: *****
  use BoldTime

  CherryClass Root:
  view:
      def index(self):
          return "<html><body>Hello, time is %s</body></hello>"%boldTime.getBoldTime()
  ```

  To compile this, just use:

  ```
  python ../cherrypy.py Hello.cpy
  ```

  Five things to note:

  – The *use* statement MUST be on the very first line of the file (don't put any comment before).
  – CherryPy will automatically create a list of dependencies, and thus read the files in order and generate the executable accordingly. If you create a loop in the dependencies, CherryPy will raise an error.

– The name of the CherryClass is *BoldTime* (with an upper case B). So is the name of the file and the name you use in the *use* statement. But when you call *boldTime.getBoldTime*, a lower b is used, because it refers to the instance of the class that is automatically created.
– You can still use regular Python *import* statements. (Either `import ...` or `from ... import ...`)
– If you have lots of modules to include with *use*, then you can split the "use" statement on several lines (but they have to be the first lines of the file). For instance, you can use:

```
****** File Root.cpy: *****
use HttpAuthenticate, CookieAuthenticate
use Mail, MaskTools

CherryClass Root:
mask:
def index(self):
OK
```

What if the modules are not in the same directory ?

Well, all you have to do is to use the *-I* option to compile the files. This allows you to specify the directories where CherryPy will look for input files. For instance, if you have the following files:

```
/dir1/Module1.cpy
/dir2/Module2.cpy
Hello.cpy (uses Module1 and Module2)
```

Then you would compile ‘Hello.cpy’ using:

```
python ../cherrypy.py -I /dir1 -I /dir2 Hello.cpy
```

By default, CherryPy will look in ‘.’, ‘../lib’ and ‘../src’

You can also set an environment variable called *CHERRYPY_HOME* that contains the name of the directory where CherryPy is installed. In this case, CherryPy will also look in ‘CHERRYPY_HOME/lib’ and ‘CHERRYPY_HOME/src’ to find the modules.

In the next chapters, we’ll learn how to use a few of CherryPy’s standard library modules.

CHAPTER

TEN

HTTP and cookie-based authentication

The two most common ways to restrict access to some parts of a website are:

- HTTP authentication: the browser opens a popup-window and prompts you for a login and password. The session information is stored inside your browser and is lost when you close all browser windows.
- Cookie-based authentication: You use a form to enter your login and password. Your session information is stored in a cookie

These techniques can be a pain to implement with some application servers. With CherryPy, they require only **THREE LINES OF CODE** !

All you have to do is use the standard modules *HttpAuthenticate* and *CookieAuthenticate*. The following is an example that uses both modules.

```
use HttpAuthenticate, CookieAuthenticate

CherryClass Root:
mask:
    def index(self):
        <html><body>
            <a py-attr="request.base+'/httpProtected/index'" href="">Click here to enter a rest
            <a py-attr="request.base+'/cookieProtected/index'" href="">Click here to enter a re
            In both cases, the login and password are "login" and "password"
        </body></html>

CherryClass HttpProtected(HttpAuthenticate):
function:
    def getPasswordListForLogin(self, login):
        # Here we define what the login and password are
        if login=='login': return ['password']
        return []
mask:
    def index(self):
        <html><body>You're in</body></html>

CherryClass CookieProtected(CookieAuthenticate):
function:
    def getPasswordListForLogin(self, login):
        # Here we define what the login and password are
        if login=='login': return ['password']
        return []
mask:
    def index(self):
        <html><body>
            You're in<br>
            Click <a href="doLogout">here</a> to log out.
        </body></html>
```

As you can see, all you have to do is to create a CherryClass that inherits from *HttpAuthenticate* or *CookieAuthenticate* and implement a function called *getPasswordListForLogin* that returns a list of matching passwords for a given login. (this allows you to keep a master key that works for all users, for instance ...)

As you can see, using these two modules is really easy.

In the next chapter, we'll see how to use another CherryPy standard module: Form

CHAPTER

ELEVEN

Handling HTML forms

Creating HTML forms can really be a pain with some application servers. Especially if you want to handle errors: if the user entered an incorrect information, the form is displayed again, all fields have kept their value, and fields that have an error stand out.

If you use CherryPy's Form module, you'll probably save yourself a lot of time (once you've understood how it works).

An exemple is provided in the demo that come's with CherryPy, and the CherryPy Library Reference has some documentation about this module.

In the next chapter, we'll learn how to configure some of CherryPy's options ...

CHAPTER

TWELVE

Configuring CherryPy

Up to now, you've always run CherryPy's server on port 8000. Well, that's nice, but how do I change that ? It's very easy: it's done through a configuration file.

12.1 Changing the port

In the 'hello/' directory, where the 'Hello.cpy' and 'HelloServer.py' files sit, create a file called 'HelloServer.cfg' with the following lines:

```
[server]
socketPort=80
```

Restart the server... It's now serving on port 80.

Some other options are available in the *[server]* section of the config file. Check out the "Deploying your website for production" chapter for more information about the different options.

12.2 Serving static content

It you want to use CherryPy to serve static content, all you have to do is add a few other lines in the configuration file:

```
[staticContent]
static=/home/remi/static
data/images=/home/remi/images
```

This means that when the browser requests the URL http://localhost/static/styleSheet.css, the server will serve the content of the file '/home/remi/static/styleSheet.css'.

When the browser requests the URL http://localhost/data/images/girl.jpg, the server will serve the content of the file '/home/remi/images/girl.jpg'

Note that if you need to server static content at the root of your website (for instance, *favicon.ico*), then you can specify the full name of the file instead of the directory, like this:

```
[staticContent]
favicon.ico=/home/remi/images/favicon.ico
```

12.3 Changing the name fo the configuration file

If you want to use a different name for the configuration file, just use the *-C* option when you start the server. For instance, if your configuration file is called '/dir1/dir2/myConfigFile.cfg', just start the server by typing:

```
python HelloServer.py -C /dir1/dir2/myConfigFile.cfg
```

CHAPTER

THIRTEEN

Using your own configuration options

You can store your own configuration options in the config file. Just add your own section and options. Then, in the code, these informations are available through the *configFile* global variable. This variable is a *ConfigParser* object. The following is an example on how to use this:

```
**** File HelloServer.cfg ****
[server]
socketPort=80

[staticContent]
static=/home/remi/static

# Here I add my own configuration options
[user]
name=Remi
[database]
login=remiLogin
password=remiPassword


*** File Hello.cpy ****
CherryClass Root:
view:
    def index(self):
        <html><body>
            Hello, <py-eval="configFile.get('user','name')"><br>
            to connect to the database, you should use:<br>
            <py-eval="'Login:%s, Password:%s'%(configFile.get('database','login'), configFile.g
        </body></html>
```

This will be rendered as:

```
<html><body>
    Hello, Remi<br>
    to connect to the database, you should use:<br>
    Login:remiLogin, Password:remiPassword
</body></html>
```

In the next chapter, we'll learn about CherryPy's special variables and special functions ...

CHAPTER

FOURTEEN

Special variables and functions

CherryPy sets and uses a few special variables and functions. They are very simple and easy to use, but also very powerful. In this chapter, we'll see what these special variables and functions are, and we'll learn how to use them in the next chapter.

14.1 Special variables

14.1.1 request

This is the most commonly used variable. It contains all the informations about the request that was sent by the client. It's a class instance that contains several member variables that are set by CherryPy for each request. The most commonly used member variables are:

- **request.headerMap**: It's a Python map containing all keys and values sent by the client in the header of the request. Note that all keys are always converted to lower case. For instance, to find out what browser the client is using, use *request.headerMap['user-agent']* (note that this information may not be sent by the client)
- **request.simpleCookie**: It's a *simpleCookie* object containing the cookies sent by the client. Note that this information is also available in *request.headerMap['cookie']*. Check out the HowTo about cookies to learn more about how to use cookie with CherryPy
- **request.base**: String containing the base URL of the website. This is equivalent to *'http://'+request.headerMap['host']*
- **request.path and request.paramMap**: The former contain the path of the page that's being requested. Leading and trailing slashes are removed (if any). The latter is a map containing a key and value for each parameter that the client sent (via GET or POST). For instance, if the URL is:

  ```
  http://localhost:8000/dir/page?key1=value1&key2=value2
  ```

  we'll have:

  ```
  request.base == 'http://localhost:8000'
  and
  request.path == 'dir/page'
  and
  request.paramMap == {'key1': 'value1', 'key2': 'value2'}
  ```

- **request.originalPath and request.originalParamMap**: These variables are a copy of *request.path* and *request.paramMap*. But we'll see in the next sections that it is possible to modify *request.path* and *request.paramMap*. In this case, *request.originalPath* and *request.originalParamMap* keep the original values.
- **request.browserUrl**: String containing the URL as it appears in the browser window
- **request.method**: String containing either *GET* or *POST*, to indicate what kind of request it was
- **request.wfile** (advanced usage only): Check out the HowTo called "How to stream uploaded files directly to disk" for more information about this

14.1.2 response

This is the second most commonly used variable (after *request*). It contains all informations about the response that will be sent back to the client. It's a class instance that contains several member variables that are set by CherryPy or by your program.

- **response.headerMap**: It's a Python map that contains all keys and values that will be sent in the header of the response. By default, CherryPy sets the following keys and values in the map:

```
"status": 200
"content-type": "text/html"
"server": "CherryPy 0.1"
"date": current date
"set-cookie": []
"content-length": 0
```

  In the next chapter, we'll learn how to use and modify these values
- **response.body**: String containing the body of the response. This variable can only be used in 3 special functions (see below)
- **response.simpleCookie**: *simpleCookie* object used to send cookies to the browser. Note that cookies can also be sent by using *response.headerMap['cookie']*. Check out the HowTo about cookies to learn more about how to use cookie with CherryPy
- **response.sendResponse and response.wfile** (advanced usage only): Used for streaming. Check out the HowTo called "How to use streaming with CherryPy" for more information.

14.2 Special functions

In your code, you can define special functions that will change the server's behavior. To define these functions, just use Python's regular syntax and define them outside all CherryClasses. When you use different modules, you can define the same function in different modules. In this case, CherryPy will just concatenate the bodies of all functions, in the same order it reads the files.

14.2.1 initRequest, initNonStaticRequest, initResponse and initNonStaticResponse

Here is the algorithm that the server uses when it receives a request:

Static content	All conten
	a. Request b. Set all r c. Call *init* d. Determi

```
e.  Read the static file and set response.headerMap values and response.body accordingly
f.  Call initResponse (which may change response.headerMap and response.body)
g.  Send the response to the browser (based on response.headerMap and response.body)
```

As you can see, *initRequest* and *initNonStaticRequest* can be used to tweak the URL or the parameters, or to perform any work that has to be done for each request.

initResponse and *initNonStaticResponse* can be used to change the response header or body, just before it is sent back to the client.

14.2.2 onError

That function is called by CherryPy when an error occured while building the page. See next section for an example.

14.2.3 initThread, initProcess

If you use a thread-pool server or process-pool server, then the corresponding special function (respectively *initThread* or *initProcess*) will be called by each newly created thread/process.

These functions can be used for instance if you want each thread/process to have its own database connection (the HowTo called "Sample deployment configuration for a real-world website" explains how to do that).

initThread takes an argument called *threadIndex* containing the index of the thread that's being created. For instance, if you create 10 threads, *threadIndex* will take values from 0 to 9.

Same thing for *initProcess* and *processIndex*

14.2.4 initProgram, initServer, initAfterBind

The code you put in *initProgram* is copied at the very beginning of the generated file, so it's the first thing that will be executed. You can use that special function if you need to run some code before the CherryClasses are instanciated. Then, the server creates all instances of the CherryClasses and then it calls the special function *initServer*. This is basically where you perform some initialization tasks if some are needed.

initAfterBind is called after the socket "bind" has been made. For instance, on Unix-based systems, you need start CherryPy as root is you want it to bind its socket to port 80. The *initAfterBind* special function can be used to change the user back to an unpriviledged user after the "bind" has been done. (the HowTo called "Sample deployment configuration for a real-world website" explains how to do that).

14.2.5 initRequestBeforeParse (advanced usage only)

This special function is called by the server when it receives a POST request, before it parses the POST data. This allows you for instance to tell the server not to parse the POST data (by setting *request.parsePostData* to *0*) and then you can parse the POST data yourself (by reading on *request.rfile*). Check out the HowTo called "How to stream uploaded files directly to disk" for more information about this

14.3 Examples

14.3.1 Playing with URLs

Let's say you want to set up a website for your customers. You want your customers to have their own URL: http://host/customerName, but the page is almost the same for each customer, so you don't want to create a method for each customer.

All you have to do is use the *initNonStaticRequest* to convert the URL http://host/customerName into http://host?customer=customerName. All that will be transparent to the user.

Just enter the following code:

```
def initNonStaticRequest():
    if request.path:
        request.paramMap['customer']=request.path
        request.path=""
CherryClass Root:
mask:
    def index(self, customer=""):
        <html><body>
            Hello, <py-eval="customer">
        </body></html>
```

And that's it !

Compile the file, start the server, and try a few URls, like http://localhost:8000/customer1 or http://localhost:8000/world

14.3.2 Sending back a redirect

To send a *redirect* to the browser, all you have to do is send back a status code of *302* (instead of 200), and set a *location* value in the response header. This can be done easily using the *response.headerMap* special variable:

```
CherryClass Root:
mask:
    def index(self):
        <html><body>
            <a href="loop">Click here to come back to this page</a>
        </body></html>
view:
    def loop(self):
        response.headerMap['status']=302
        response.headerMap['location']=request.base
        return "" # A view should always return a string
```

14.3.3 Adding timing information to each page

In this example, we'll add one line at the end of each page that's served by the server. This line will contain the time it took to build the page. Of course, we only want this line for dynamic HTML pages.

All we have to do is use *initNonStaticRequest* to store the start time, and use *initNonStaticResponse* to add the line containing the build time.

Here is the code:

```
import time
def initNonStaticRequest():
    request.startTime=time.time()
def initNonStaticResponse():
    if response.headerMap['content-type']=='text/html':
        response.body+='<br>Time: %.04fs'%(time.time()-request.startTime)
CherryClass Root:
mask:
    def index(self):
        <html><body>
            Hello, world
        </body></html>
```

And voila

14.3.4 Customizing the error message

This is done through the *onError* special function. Just use *response.headerMap* and *response.body* to do what you want.

The following example shows how to set it up so it sends an email with the error everytime an error occurs:

```
use Mail

def onError():
    # Get the error in a string
    import traceback, StringIO
    bodyFile=StringIO.StringIO()
    traceback.print_exc(file=bodyFile)
    errorBody=bodyFile.getvalue()
    bodyFile.close()
    # Send an email with the error
    myMail.sendMail("erreur@site.com", "webmaster@site.com", "", "text/plain", "An error occure
    # Set the body of the response
    response.body="<html><body><br><br><center>"
    response.body+="Sorry, an error occured<br>"
    response.body+="An email has been sent to the webmaster"
    response.body+="</center></body></html>"


CherryClass MyMail(Mail):
function:
    def __init__(self):
        self.smtpServer='smtp.site.com'

CherryClass Root:
mask:
    def index(self):
        <html><body>
            <a py-attr="request.base+'/generateError'" href="">Click here to generate an error<
        </body></html>
    def generateError(self):
        <html><body>
            You'll never see this: <py-eval="1/0">
        </body></html>
```

This example also shows you how to use the *Mail* standard module that comes with CherryPy.

CHAPTER

FIFTEEN

Deploying your website for production

So there you are, with a nice big website you've spent weeks working on. And it's finally ready for the world to use it !

But you still have to decide how you will **deploy** it, which means: how will you set it up on the production machine(s).

15.1 Choosing you deployment configuration

When you are developing your website, you're usually the only one (maybe with a few other developers) accessing the website, so it doesn't need to be fast and robust. But on the production website, many people (if you're lucky) will access your website. This means that you have to choose the proper CherryPy configuration in order to provide a fast/reliable service to your users.

Criteria to help you choose your configuration include:

- **What does you hosting provider let you do ?** If you're on a shared machine, you might not be able to do what you want. For instance, you may only be able to use CGI, and your hosting provider may only provide virtual hosting, behind Apache.
- **How much traffic do you plan to get ?** Do you plan to have only a few hundreds users per day or several tens of thousands ?
- **How many machines/processors do you have ? (ie: how much money do you have)** If you plan to have a lot of traffic, then you might have to use several machines/processors (which means higher cost).
- **Will there be a webmaster looking after the website ?** If you don't have anyone looking after the website at all time, you might want the website to restart automatically in case of a crash.

Note that there is a HowTo called "Sample deployment configuration for a real-world website" that shows a full sample configuration that is recommended for most websites.

15.1.1 Should I use the CherryPy HTTP server directly or behind another webserver like Apache ?

The first decision to make is whether to use the CherryPy HTTP server directly or behind another webserver like Apache. Here is a list of advantages for each method:

Use it directly

- Is faster and uses less ressources (no Apache processes and no need to talk between Apache and CherryPy)

- Is easier to set up

Use it behind Apache

- Might be faster for serving static content (like images)
- Hosting provider might force you to use Apache

Once you've decided if you wanted to use CherryPy directly or behind another webserver, you still have to decide among several configurations...

15.1.2 Options for deploying CherryPy directly

The following subsections show you what the different options are and what the advantages/drawbacks are:

Single thread/process

Explaination: This means that the CherryPy HTTP server will run in one single thread/process. While it is handling a request, no other request can connect to it during that time.

Advantages:

- Fast for each request (no need to create a thread/process for each request)

Drawbacks:

- Cannot handle concurrent requests

Conclusion: This method is the default configuration and it works fine for development, but it should be banned for production if you might have several users accessing your website at the same time.

Forking server

Explaination: This means that the CherryPy HTTP server will create a new process to handle each request. After the response is sent back, the process is destroyed.

Advantages:

- Can handle multiple requests at a time
- On a multiprocessor machine, a forking server will take advantage of the several processors

Drawbacks:

- Might be expansive to create a new process for each request (especially if requests come in very fast)
- Forking doesn't work on Windows
- Cannot easily use sessions as session data cannot be easily shared among processes

Conclusion: This method can be used on non-Windows machines if the website's traffic isn't too high.

Threading server

Explaination: This means that the CherryPy HTTP server will create a new thread to handle each request. After the response is sent back, the thread is destroyed.

Advantages:

- Can handle multiple requests at a time
- Works on all platforms (including Windows)

Drawbacks:

- Might be expansive to create a new thread for each request (although less expensive than processes), especially if requests come in very fast
- On a multiprocessor machine, a threading server will **not** take advantage of the several processors (due to the Python global interpreter lock)

Conclusion: This method can be used if the website's traffic isn't too high.

Process pool

Explaination: This means that the CherryPy HTTP server will create a fixed number of processes at startup, and these processes will remain all the time. If one process if busy handling a request and another request comes in, then the next process will step up and handle it.

Advantages:

- Can handle multiple requests at a time
- Fast because we don't need to create a thread or process for each request
- Takes advantage of multi-processor machines

Drawbacks:

- Doesn't work on Windows
- Cannot easily use sessions as session data cannot be easily shared among processes

Conclusion: This method works well on non-Windows machines, as long as you don't have hundreds of concurrent users.

Thread pool

Explaination: This means that the CherryPy HTTP server will create a fixed number of threads at startup, and these threads will remain all the time. If one thread if busy handling a request and another request comes in, then the next thread will step up and handle it.

Advantages:

- Can handle multiple requests at a time
- Fast because we don't need to create a thread or process for each request

Drawbacks:

- Doesn't take advantage of multi-processor machines.
- Number of threads doesn't increase if we have more concurrent users.

Conclusion: This method works very well and it is the recommended set-up in many cases (as long as you don't have hundreds of concurrent users).

Other alternatives

If you **really** have a lot of traffic and the previous methods are not enough or you can not use them (if you're on Windows for instance), then you can use generic load-balancing. There is a HowTo in the documentation about it.

15.1.3 Options for deploying CherryPy behind another webserver

All the configurations described in the previous section are also available when deploying CherryPy behind another webserver. The third-party webserver will generally be multi-threaded or multi-processes. There is a HowTo in the documentation that explains how to set this up.

15.2 Configuration file options

Here is the list of the configuration file options that are used to specify how the CherryPy server will be deployed. All these options are used within the *[server]* section of the configuration file (cf Chapter "Configuring CherryPy").

- **socketPort**: This indicates which port the server should listen to. Example:

```
[server]
socketPort=80
```

- **socketHost**: This indicates which address the server should bind to (the default is localhost). Example:

```
[server]
socketHost=192.168.0.23
```

- **socketFile**: This is only used on Unix, if you want to use an AF_UNIX socket instead of a regular AF_INET socket. Example:

```
[server]
socketFile=/tmp/mySocket.soc
```

- **forking**: Set this to 1 if you want a forking server. Example:

```
[server]
socketPort=80
forking=1
```

- **threading**: Set this to 1 if you want a threading server. Example:

```
[server]
socketPort=80
threading=1
```

- **processPool**: Set this to n (n>1) if you want to have n processes created at startup. Example:

```
[server]
socketPort=80
processPool=10
```

- **threadPool**: Set this to n (n>1) if you want to have n threads created at startup. Example:

```
[server]
socketPort=80
threadPool=10
```

- **reverseDNS**: Set this to 1 if you want to enable reverse DNS (this way the full name of the domain name for the clients will be written to the logs). The default is 0. Example:

```
[server]
socketPort=80
reverseDNS=1
```

- **socketQueueSize**: Size of the socket queue (this value will be passed to the listen() function). The default is 5. Example:

```
[server]
socketPort=80
socketQueueSize=5
```

- **sslKeyFile and sslCertificateFile**: This is used to have an SSL server. There is a HowTo in the documentation about it.
- **xmlRpc**: This is used to have an XML-RPC server. There is a HowTo in the documentation about it.

Some of these options obviously cannot be used together because they conflict:

- **socketFile** and **socketPort** obviously conflict with each other
- **threading**, **forking**, **processPool** and **threadPool** obviously conflict with each other

CHAPTER

SIXTEEN

And now what ?

This tutorial should be enough to understand how CherryPy works and to develop some real-world websites with it.

You can also check out the rest of the documentation (HowTos, library module reference ...) for more advanced features.

If you need some help, just post a message to the mailing list and we'll be happy to help you.

We hope you'll have as much fun using CherryPy as we do.

CHAPTER

SEVENTEEN

History and License

17.1 License

CherryPy is released under the GPL license.