CherryPy HowTo

Release 0.10

Remi Delon

March 19, 2004

Email: remi@cherrypy.org

CONTENTS

CHAPTER

ONE

How to serve gzip-compressed pages with CherryPy

Most browser support gzip-compression. If you have large HTML pages (40K or more), compressing them can really make a big difference (a 50K page might only be 5-6K when it's compressed).

Doing that with CherryPy is really easy. It only takes a few lines of code and uses the *initNonStaticResponse* special function:

```
import gzip, cStringIO

def initNonStaticResponse():
    # Only compress the page if the client said it accepted it
    if request.headerMap.get('accept-encoding', '').find('gzip')!=-1:
        # Compress page
        zbuf=cStringIO.StringIO()
        zfile=gzip.GzipFile(mode='wb', fileobj=zbuf, compresslevel=9)
        zfile.write(response.body)
        zfile.close()
        response.body=zbuf.getvalue()
        response.headerMap['content-encoding']='gzip'
```

CHAPTER

TWO

How to run a CherryPy server behind Apache

2.1 Introduction

CherryPy's built-in HTTP server is now pretty robust, but some people might still want to run CherryPy behind Apache.

Whether to run CherryPy exposed or behind Apache depends on many criteria so this question is out of the scope of this HowTo.

This HowTo will show you how to run CherryPy behind Apache, but you can probably adapt it for any other webserver.

There are several ways to run CherryPy behind Apache:

2.2 Using persistent CGI

The way it works is very easy to understand. We use a small cgi script called **cherrypcgi.cgi** as a link between Apache and CherryPy. When someone requests a page, Apache invokes this cgi script. The script then connects to the CherryPy server, gets the page and returns it to Apache.

Of course, it takes a little extra time because a new process is created everytime the script is called. And also because the request and the response go through the script instead of going directly from Apache to CherryPy and back.

But that extra time is really small, so it's not a big problem. (and if you have a really high traffic website, you can use load balancing anyway :-)

2.2.1 Example

The cgi script uses the HTTP protocol to talk to the CherryPy server. Therefore, the CherryPy server doesn't make any difference between talking to a browser or talking to the cgi script.

This means that you can just start it normally.

If you're on Unix, it's probably better to run the CherryPy server on an AF_UNIX socket. This is what we will do in this example.

To do so, edit the configuration file of the CherryPyServer and enter the following lines:

```
[server]
socketFile=socketFile.soc
```

This means that the CherryPy server will listen on that AF_UNIX socket instead of a regular socket port.

Now, just edit the 'cherrypcgi.cgi' file provided with the distribution and modify the line that says:

```
socketFile='put your socket file here'
```

Put the name of the AF_UNIX socket where the CherryPy server is listening.

The last thing to do is to configure Apache so it will call the cgi script for each request: In Apache's configuration file ('commonhttpd.conf' for instance), add the following lines (of course, you have to adapt the path of cherrypcgi.cgi):

```
RewriteEngine on
RewriteRule ^(.*) /home/cherrypy/cherrypcgi.cgi$1 [e=HTTP_CGI_AUTHORIZATION:%1,t=application/x-
```

And that's it ! Start Apache, start the CherryPy server and it should work.

It is also possible to tweak cherrypcgi.cgi so it automatically restarts the CherryPy server if it ever goes down. There will be another HowTo on this.

2.3 Using FastCGI

FastCgi works very much like persistent CGI, except the cgi script is constantly running. This means that no process has to be created for each request, which saves a lot of time !

The current implementation of FastCGI in CherryPy is not optimized because the FastCGI script is a standalone script that connects to the CherryPy backend, instead of being directly integrated in the backend. But this method is still a lot faster than persistent CGI.

The FastCGI script is called **cherryfcgi.cgi**

2.3.1 Example

The FastCGI script uses the HTTP protocol to talk to the CherryPy server. Therefore, the CherryPy server doesn't make any difference between talking to a browser or talking to the FastCGI script.

This means that you can just start it normally.

If you're on Unix, it's probably better to run the CherryPy server on an AF_UNIX socket. This is what we will do in this example.

To do so, edit the configuration file of the CherryPyServer and enter the following lines:

```
[server]
socketFile=socketFile.soc
```

This means that the CherryPy server will listen on that AF_UNIX socket instead of a regular socket port.

Now, just edit the 'cherryfcgi.cgi' file provided with the distribution and modify the line that says:

```
socketFile='put your socket file here'
```

Put the name of the AF_UNIX socket where the CherryPy server is listening.

The last thing to do is to configure Apache so it will connect to the FastCGI script for each request: In Apache's configuration file ('commonhttpd.conf' for instance), add the following lines (of course, you have to adapt the path of cherrypcgi.cgi):

```
SetHandler fastcgi-script
RewriteEngine on
RewriteRule ^(.*) /home/cherrypy/cherryfcgi.cgi/$1 [L]
```

You also need to make sure that Apache loads the FastCGI module. The following lines should be somewhere in your Apache configuration files:

```
LoadModule fastcgi_module    modules/mod_fastcgi.so
AddModule mod_fastcgi.c
```

And that's it ! Start Apache, start the CherryPy server and it should work.

See Also:

http://www.fastcgi.com
: For more information on FastCGI

2.4 Using mod_rewrite

It's easy to configure Apache so it just passes all requests to the CherryPy server, reads the response and passes the response to the client.

This method is a bit faster than persistent CGI because no CGI process needs to be created for each request.

2.4.1 Example

In this example, we'll assume that the web site is www.cherrypy.org and that the CherryPy server is running on the same machine as Apache, on the port 8000. Of course, you'll have to adapt it for your own configuration.

Configuring Apache is very easy:

```
RewriteEngine on
RewriteRule ^(.*) http://localhost:8000$1 [p]
```

The last thing we have to do is tell CherryPy that it's serving pages for www.cherrypy.org (and not localhost). All it takes is 3 lines of code in the *initRequest* special function:

```
def initRequest():
    request.headerMap['host']='www.cherrypy.org'
    request.base='http://www.cherrypy.org'
    request.browserUrl=request.browserUrl.replace('http://localhost:8000', 'http://www.cherrypy
```

And voila !

Note, this can also be done with mod_proxy instead of mod_rewrite.

Many thanks to Andreas Kostyrka for this tip.

CHAPTER

THREE

How to connect a CherryPy server to a database

Many databases already have a corresponding Python module: Oracle, Sybase, MySql, PostgreSql, ...

Accessing these databases from a CherryPy server is really easy. All you have to do is use the *initServer* function to connect to the database.

For instance, you can connect to a MySql database using the following code (it uses the MySQLdb Python module):

```
import MySQLdb
def initServer():
    global dbConnection
    dbConnection=MySQLdb.connect(host, user, passwd, schema)
```

The following code is an example of running a query and using the result to build a page:

```
CherryClass Root:
view:
    def index(self):
        # Run the MySql query
        cursor=dbConnection.cursor()
        cursor.execute("select count(*) from user")
        result=cursor.fetchall()
        cursor.close()

        page="<html><body>"
        nbUser=result[0][0]
        page+="There are %s users in the database"%nbUser
        page+="</body></html>
        return page
```

This example can easily be adapted for any kind of database. Just find the appropriate Python module for the database and read its documentation.

CHAPTER

FOUR

How to use load-balancing for your web site

4.1 Introduction

Having a lot of traffic on their web site is what most webmasters dream about ...

Unfortunately, it also comes with its share of problems. Basically, if requests come in faster than your server can handle them, you're screwed :-)

One way to deal with that is to use load-balancing. This basically means that several threads or processes will be serving the pages. It is only efficient if you have several machines or if your machine has several processors. (otherwise, the same processor will just run several threads or processes, but the overall speed will be the same).

Of course, this means that you have to take care of the data sharing between the threads/processes. One way to do that is to use a database where you store all the data that needs to be shared amongst the processes. All processes read and write to the same database, insuring that they all have the same informations.

There are two ways to do load-balancing with CherryPy. One of them is easier to set up, but only applies to multi-processor machines running a Unix-based OS.

4.2 Generic load-balancing method

Since a CherryPy server is a self-contained process that includes everything to run the web site, it is easy to start as many processes as you want (on the same machine or on different machines). If you start several processes on the same machine, you just have to use a different configuration file for each of them to specify a different port each time.

Then all you have to do is use a simple load-balancer (there are many of those out there) to redirect the requests to your CherryPy servers.

Let's take an example:

- You have 3 machines, called host1, host2 and host3
- host2 and host3 are single-proc, and host1 is dual-proc.
- For the load-balancer, we'll use **balance**, an easy to use, lightweight, open source load balancer available from http://balance.sourceforge.net.
- On host1, we'll run 2 CherryPy servers (since it's dual-proc) plus the load-balancer
- On host2 and host3, we'll only run one CherryPy server.

- **balance** will listen on port 80. The CherryPy servers on host1 will listen on ports 8080 and 8081. The CherryPy servers on host2 and host3 will listen on port 8080.
- Let's assume that your CherryPy server file is called 'MyServer.py'

Here is what we have to do:

- Copy 'MyServer.py' on host1, host2 and host3. (alternatively, you can export the file system from one machine to the others, to deal with only one copy of the file)
- On host1, host2 and host3, create a configuration file called 'MyServer8080.cfg' containing the following:

```
[server]
socketPort=8080
```

  On host1, create another configuration file called 'MyServer8081.cfg' containing the following:

```
[server]
socketPort=8081
```

- On host1, host2 and host3, start the servers that listen on port 8080:

```
[host1] %  python MyServer.py -C MyServer8080.cfg
[host2] %  python MyServer.py -C MyServer8080.cfg
[host3] %  python MyServer.py -C MyServer8080.cfg
```

- On host1, start the second server that listens on port 8081:

```
[host1] %  python MyServer.py -C MyServer8081.cfg
```

- Now all you have to do is start the load balancer and indicate what the available CherryPy servers are:

```
[host1] % /usr/sbin/balance 80 host1:8080 host1:8081 host2:8080 host3:8080
```

And voila !

4.3 Multi-processor, unix-based machine

The method described in the previous section works in all cases, but if you have a unix-based machine with multiple processors, there is another method for load-balancing with CherryPy.

The trick is to create the socket where the CherryPy server will listen, and then do a **fork()**. This way, we'll have multiple processes listening on the same socket. When one process is busy building a page, the next one will be listening on the socket and thus serving a request that might come in.

This feature is built in. All you have to do to use it is to use the **fixedNumberOfProcesses** option in the configuration file (in the *[server]* section):

```
[server]
socketPort=80
fixedNumberOfProcesses=3
```

And that's it.

Note: This method is only useful if your machine has several processors. If not, then the processor will run multiple CherryPy processes, but the overall speed won't be improved.

CHAPTER

FIVE

How to compile your code in debug mode

One way to debug your code is to add `print` statements in your code. The output will show up in the window where the CherryPy server runs. However, this method may not always work (for instance, if the server is running as a daemon on a production machine and you don't have access to its standard output). Plus switching from debug mode to non-debug mode is not easy (you have to comment out all the `print` statements ...).

Another way is to save this output in a log file. You can redirect the standard output from the server, or you can write in the log file directly from your code. CherryPy provides one special function, one CGTL tag and one CHTL tag to do that.

The special function is called *debug* and the tag is called *py-debug*. They are used like this:

```
debug('i=%s'%i)
<py-debug="'i=%s'%i">
<div py-debug="'i=%s'%i"></div>
```

All these statements have the same effect: add a line in the log file.

If the name of the CherryPy server is 'RootServer.py', the name of the log file is 'RootServer.log'.

Debugging can be easily turned on or off at compile time, using the **-D** option of the compiler. If **-D** is specified, then debugging is turned on:

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

If **-D** is omitted, debugging is turned off.

CherryPy uses a special global variable called *_debug* that can be accessed from anywhere in your code. The variable is 1 if debugging is on, 0 otherwise.

CHAPTER

SIX

How to use the hotReload feature of CherryPy

6.1 Introduction

The development cycle of a CherryPy web site is the same as the development cycle of a regular software:

- Write source files
- Compile them
- If compiler generated some errors, fix source files and try to compile again
- Once compilation was OK, stop running executable and restart the newly generated executable
- If we find some bugs or we want some new features, modify the source files and go back to the compilation stage

The compilation process is usually pretty fast (if you have a "decent" computer).

But what if the CherryPy server has a long initialization process (maybe it needs to connect to several ressources, like databases, or it needs to precompute some data). All this work is usually done in the *initServer* special function. If this initialization takes a lot of time, having to stop and restart the server everytime you make a change is really a loss of time while developing your web site.

This is where the *hotReload* feature comes in. It allows you to refresh your code without having to stop the server.

6.2 How does it work ?

When it receives a *hotReload* request, the server will just read its source code again, updating all the CherryClasses code and recreating new CherryClasses instances with the new code. It will not execute *initServer* again. Instead, it will execute *hotReloadInitServer* if any.

6.3 How to use it ?

The *hotReload* feature is enabled in the executable when the file is compiled with the *-D* flag (debug mode).

To send a *hotReload* request to a running CherryPy server, just use the little script called ‘cherryhotreload.py’ provided with the distribution.

Let’s take a practical example that shows how to use the *hotReload* feature:

- You write a first version of your source file called 'Root.cpy'
- You compile your source file in debug mode:

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

- In a window, you start the CherryPy server:

```
python RootServer.py
```

- Now you have a running CherryPy server
- You realize that you want to change something in your code
- Just update your source file and recompile it (in another window, since the server is still running in the first window)
- Send the *hotReload* request to the running server to tell it to refresh its code:

```
python cherryhotreload.py localhost:8080
```

  (This assumes that the server is running on localhost:8080. Of course, you have to adapt it for your own configuration)

- In the window where the server is running, you see a message saying that it has performed the *hotReload*

Warning: After a *hotReload*, the line numbers that the python interpreter might give you in case of an error may not match your source file. This happens if you add or remove lines of code before the *hotReload*

CHAPTER

SEVEN

How to use caching

7.1 Introduction to caching

If you have a web site where pages take a long time to build but don't change too often, then caching is just the tool you need to speed up your web site.

Here is how caching works:

- The first time someone requests a page, the server builds the page normally and returns the page to the client
- The server also saves a copy of the page "somewhere" for later retrieval. It also sets a delay for how long this copy can be used.
- The next time someone requests the same page, the server uses the copy it made instead of rebuilding the page.
- When the copy expires, the server builds a new version of the page.

7.2 Caching with CherryPy

7.2.1 Where are pages stored ?

With CherryPy, all pages are saved in memory. This allows for maximum speed, but it also means that the size of your process will grow really fast if you cache lots of big pages. To avoid that, CherryPy provides a way to "purge" old pages that are in the cache.

7.2.2 How does it know if a page is already in the cache ?

The page that a server returns to a client usually depends on 3 parameters:

- The URL that the client requested
- The parameters that the client sent (via a GET or a POST)
- The cookies that the client sent

CherryPy lets you define what your cache "key" will be. Two requests that have the same cache key will receive the same response from the server.

For instance, if your pages don't depend on cookies, your cache key could just be *request.browserUrl*. But if your pages depend on cookies, your cache key can be *request.browserUrl+str(request.simpleCookie)*.

7.2.3 How do I control which pages I want to cache or not ?

It is very easy. All you have to do is use *initRequest* or *initNonStaticRequest* to set a couple of special variables if you want to use caching:

- *request.cacheKey*: this is the cache key as described in the previous section
- *request.cacheExpire*: This is the time the cached version of the page will expire

Let's take an example:

- You have a web site with 3 main parts: /part1, /part2 and /part3
- Pages under /part1 are complicated to build, and the pages might change every 30 minutes: we'll use caching on this part and the caching delay will be set to 10 minutes.
- Pages under /part2 are complicated to build, and the pages might change every day: we'll use caching on this part and the caching delay will be set to 12 hours.
- Pages under /part3 are very fast to build, and not many people visit this area: we won't use caching on this part
- Pages only depend on the URL, not on cookies: we'll use the URL as the cache key

Here is what the *initNonStaticRequest* special function will look like:

```
import time

def initNonStaticRequest():
    if request.path.find('part1')==0:
        request.cacheKey=request.browserUrl
        request.cacheExpire=time.time()+30*60 # 30 minutes
    elif request.path.find('part2')==0:
        request.cacheKey=request.browserUrl
        request.cacheExpire=time.time()+12*60*60 # 12 hours
```

That's all ! Just restart your server and the pages will be cached

7.2.4 How do I control when the cache is purged ?

Since all cached pages are stored in memory, the memory usage of the server will grow really fast if it has to cache lots of different pages (or, to be more precise, "different cache keys"), especially if these pages are big. For this reason, you should only cache pages for which it will really make a speed difference (good candidates are pages that are complicated to build but that don't change too often).

If all your pages in the cache are requested very often, there isn't much you can do to lower the memory usage.

But if some of them are only requested once in a while, then it might be worth it to remove them from the cache when they haven't been requested for a long time. This will free up some memory.

To control how often CherryPy will purge the cache, use a parameter called *flushCacheDelay* in the section *cache* of the config file:

```
[cache]
flushCacheDelay=10 # In minutes
```

The default value for the *flushCachDelay* is zero, which means that the cache will never be flushed.

Note: This parameter is only useful in very specific cases, so you probably don’t need to use it ...

CHAPTER

EIGHT

How can webdesigners and webdevelopers collaborate on a CherryPy project

8.1 Introduction

The two common kinds of people involved in a website development are designers and developers. Designers typically take care of the presentation of the pages, and developers take care of the content and the logic behind the web site.

Designers and developers don't use the same tools: designers usually use an HTML editor (for instance, Dreamweaver or Amaya) and developers use a text editor and CherryPy.

8.2 How can they collaborate ?

It's very easy:

- Designers create template files with their HTML editor
- Developers edit the template files and add some dynamic data if needed (using CHTL)
- Since CGTL is HTML-editor-safe, designers can re-edit the template files and add new HTML code, without losing the dynamic information.
- Developers include the template files in their CherryPy source code using the <py-include> tag in their masks:

```
<div py-include="header.html">header</div>   (CHTL form)
or
<py-include="header.html"> (CGTL form)
```

The template file will be included in the code by the CherryPy preprocessor (very much like an "#include" directive in C). To find the template files, the preprocessor will look in the same directories as for the other source files ('/lib', '/src', directories specified with the -I flag, etc.)

8.3 Example

In this example, we'll build a small website with only 2 pages. Each page will be made of a header, a body and a footer. The header and the footer will be the same on both pages, except they will have a "you are here" label.

Let's go step by step

- The webdesigners create 4 files with their favorite HTML editor:

```
******************
* File header.html
******************
<html><body>
Welcome to the CherryPy py-include demo.<br>
You are here: home<br><br>

******************
* File footer.html
******************
<br><br><small>Don't forget to eat cherry pie every day</small>
</body></html>

******************
* File page1.html
******************
Header

Welcome to page1.<br>
Click <a href="page2">here</a> to go to page 2.

Footer

******************
* File page2.html
******************
Header

Welcome to page2.<br>
Click <a href="page1">here</a> to go to page 1

Footer
```

  Nothing too fancy so far. We just have a header, a footer, and two bodies. Everything is static.

- At this point, the webdevelopers will edit the html files and add some dynamic information:
  - In header.html, change 'home' into ' home '
  - In page1.html and page2.html, change 'Header' into ' Header ' and 'Footer' into ' Footer '
- Now the templates are ready, and the webdevelopers can write their CherryPy source code:

```
CherryClass Root:
mask:
    def page1(self, youAreHereLabel='page1'):
        <py-include="page1.html">
    def page2(self, youAreHereLabel='page2'):
        <py-include="page2.html">
```

We now have a working version of the web site: just compile the CherryPy source file, run the server and test it.

- Now, let's assume that the designers want to change something in the look of the page: they want the "CherryPy" word in the header to be displayed in bold . To do so, all they have to do is edit the "header.html" file and put the "CherryPy" word in bold. The HTML editor should leave the ' home untouched. (it keeps the dynamic information).

CHAPTER

NINE

How to use SSL with CherryPy

9.1 Introduction

There are several ways to make an SSL CherryPy website:

- Either by running the CherryPy server behind Apache and by configuring Apache to handle SSL
- Or by running the CherryPy server directly and by configuring CherryPy to handle SSL

Only the second option is covered in this HowTo.

9.2 Prerequisite

CherryPy's SSL support is based on the PyOpenSSL package. You can get it at http://pyopenssl.sourceforge.net. I recommend using version 0.5.1 or higher of PyOpenSSL. Problems have been reported with earlier versions on some platforms.

You can test your installation by firing up the Python interpreter and typing:

```
>>> import OpenSSL
>>>
```

If this doesn't work for you, then PyOpenSSL is not correctly installed on your system.

9.3 Configuring the CherryPy server

Once you have PyOpenSSL installed, all you have to do is add 2 lines in your CherryPy config file, in the *server* section:

```
sslKeyFile=/path/to/ssl/key/file
sslCertificateFile=/path/to/ssl/certificate/file
```

And that's it !

A "real-world" sample config file for an SSL site might be:

```
[server]
socketPort=443
sslKeyFile=server.pkey
sslCertificateFile=server.cert
```

CHAPTER

TEN

How to use XML/XSL with CherryPy

10.1 Introduction

With CherryPy, you can use python standard datatypes (lists, dictionaries, ...) to store your data, and then you can use masks (written in CHTL) to convert that data into HTML.

However, this can also be done with XML and XSL. I personally think that using python types directly, and then writing CHTL masks is much easier and straightforward than XML/XSL, but there might be several reasons why you'd want to use XML/XSL:

- You're getting your data from some other application and that data comes in XML (which is likely to happen, since XML/XSL is now a standard).
- Your website is a multi-million contract with a big company, and they want to hear words like "XML" or "XSL", because it makes them feel safe ...
- Your boss will let you use CherryPy, but only if you use XML/XSL, because it makes him feel safe ...

10.2 Prerequisite

This HowTo uses the 4Suite module developed by Fourthought, Inc. I tested it with version 0.12.0a1 of 4Suite and version 2.1 of python, but other combinations would also probably work.

Also, several other XML/XSL modules are available for Python, and using them is also probably very easy.

Start by downloading 4Suite from http://4suite.org and installing it on your machine.

You can test your installation by firing up the Python interpreter and typing:

```
>>> from Ft.Xml.Xslt.Processor import Processor
::: Using pDomlette
>>>
```

If this doesn't work for you, then 4Suite is not correctly installed on your system.

10.3 Using the XML/XSL package from CherryPy

Once you have 4Suite installed, is is very easy to use it from CherryPy. You can use regular masks to write your XSL stylesheets, and you have several ways to generate your XML (using a function, a view or even a mask).

The following example is a simple example that demonstrates a basic XML/XSL transformation:

```
from Ft.Xml.Xslt.Processor import Processor

CherryClass XslTransform:
function:
    def transform(self, xslStylesheet, xmlInput):
        processor = Processor()
        processor.appendStylesheetString(xslStylesheet)
        return processor.runString(xmlInput, 0, {})

CherryClass Root:
view:
    def index(self):
        return xslTransform.transform(self.xslStylesheet(), self.xmlInput())
mask:
    def xslStylesheet(self):
        <?xml version="1.0" encoding="ISO-8859-1"?>

        <xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

        <xsl:template match="/">
            <html><body>
                <h2>My CD Collection</h2>
                <table border="1">
                    <tr bgcolor="#9acd32">
                        <th align="left">Title</th>
                        <th align="left">Artist</th>
                    </tr>
                    <xsl:for-each select="catalog/cd">
                        <tr>
                            <td><xsl:value-of select="title"/></td>
                            <td><xsl:value-of select="artist"/></td>
                        </tr>
                    </xsl:for-each>
                </table>
            </body></html>
        </xsl:template>

        </xsl:stylesheet>
    def xmlInput(self):
        <?xml version="1.0" encoding="ISO-8859-1"?>
        <catalog>
            <cd>
                <title>Empire Burlesque</title>
                <artist>Bob Dylan</artist>
            </cd>
            <cd>
                <title>Hide your heart</title>
                <artist>Bonnie Tyler</artist>
            </cd>
        </catalog>
```

How does it work ?

- *xslStylesheet* is a mask containing the XSL stylesheet

- *xmlInput* is a mask containing the XML input. In real life, this XML would probably be generated from a function, based on data coming from a database or some other ressource.
- The *index* view just says to apply this XSL stylesheet to that XML input.
- The *XslTransform* CherryClass is just used to implement the *transform* function, to avoid having to repeat these 3 lines of code for each transformation.

Here is the HTML document generated in this example:

```
<html>
  <body>
    <h2>My CD Collection</h2>
    <table border='1'>
      <tr bgcolor='#9acd32'>
        <th align='left'>Title</th>
        <th align='left'>Artist</th>
      </tr>
      <tr>
        <td>Empire Burlesque</td>
        <td>Bob Dylan</td>
      </tr>
      <tr>
        <td>Hide your heart</td>
        <td>Bonnie Tyler</td>
      </tr>
    </table>
  </body>
</html>
```

PS: Note that it seems that the 4Suite API has changed again in the latest releases, so you might have to tweak the above code if you're using a recent release.

CHAPTER

ELEVEN

How to use AOP (Aspect Oriented Programing) with CherryPy

11.1 Introduction

This HowTo is intended to people who are already familiar with OOP, but who are not necessarily familiar with AOP. If this is your case, check out http://aosd.net for an introduction to AOP.

CherryPy only implements a few of AOP concepts. I don't have a PHD in AOP and it is not my goal to make CherryPy an AOP reference :-)

Instead, I implemented a few features that I thought might be useful (and they already are, since both HttpAuthenticate and CookieAuthenticate modules use AOP features).

11.2 Basic example

On most websites a common header and footer is used for all (or most) pages of the site. In CherryPy, you can easily implement that using a "classic" object oriented approach: first you define a class that contains the header and the footer that you want to use for all your pages.

```
CherryClass NormalHeaderAndFooter:
mask:
    def header(self):
        <html><body>I'm the header<br><br>
    def footer(self):
        <br><br>I'm the footer</body></html>
```

Then, all you have to do is derive your CherryClasses from *NormalHeaderAndFooter* and call *header* and *footer* in each of your masks or views:

```
CherryClass Root(NormalHeaderAndFooter):
mask:
    def index(self):
        <py-eval="self.header()">
            Hello, world !
        <py-eval="self.footer()">
    def otherPage(self):
        <py-eval="self.header()">
            I love cherry pie !
        <py-eval="self.footer()">
```

That's the "classic" object oriented approach. It's pretty good, but if we look at it, we'll realize that we have to repeat *<py-eval="self.header()">* and *<py-eval="self.footer()">* for each mask and view, and this has several drawbacks:

- First of all, I **never** like to repeat code. Whenever I do it, I feel like there must be something I'm doing wrong and my code could certainly be improved !
- For a real life website, we probably would have many CherryClasses that inherit from *NormalHeaderAndFooter*, and all these CherryClasses would have many masks that all make the same call to *header* and *footer*. What if we want to change the name of the *header* method ? In that case, we would have to go through all the masks of all the CherryClasses to change the line !
- In this trivial example, we only have 2 lines of code to repeat, but what if we had other things to do for each page (like log the request in a database or check that the user is authenticated) ? We could end up repeating tens of lines of code for each method !

That's where AOP comes in ... Instead of just declaring "normal" *header* and *footer* methods, we'll declare "aspect" methods. This basically adds another information which is: "include my code at the beginning or at the end of each of the methods of all derived classes".

Therefore, the implementation of our simple example using AOP would be:

```
CherryClass NormalHeaderAndFooter:
aspect:
    (1) start:
        _page.append("<html><body>I'm the header<br><br>")
    (1) end:
        _page.append("<br><br>I'm the footer</body></html>")

CherryClass Root(NormalHeaderAndFooter):
mask:
    def index(self):
            Hello, world !
    def otherPage(self):
            I love cherry pie !
```

[1]

See ... we got rid of the *<py-eval="self.header()">* and *<py-eval="self.footer()">* lines ...

Let's see what we can notice from looking at the code:

- In CherryClass *NormalHeaderAndFooter*, the "mask" section has disappeared and has been replaced by a new "aspect" section.

[1] This sample code requires CherryPy-0.7 or later to work

- In this "aspect" section, the line where we normally declare methods (using the "def" keyword) has been replaced by a special line that contains "start" or "end" (we'll explain the syntax later).
- The lines that normally contain the body of the function are still there, and they're written using regular Python.
- *<py-eval="self.header()">* and *<py-eval="self.footer()">* have disappeared from the methods of *Root*. *Root* is still derived from *NormalHeaderAndFooter*.

Here is how it works:

The header for an aspect method has two parts:

```
(condition) startOrEnd:
```

startOrEnd can be either the "start" or the "end" keyword. This indicates whether the code should be appended at the start or at the end of the methods.

condition is a python expression used to indicated which method the aspect should apply to. If the condition is always true (as in "1"), then it means that the aspect will apply to all methods of the derived CherryClasses. The python expression can use a special variable called *method* that contains the following member variables:

- *name*: contains the name of the method
- *type*: contains the type of the method ("mask", "view", "variable", ...)
- *isHidden*: is true if the mask or view is hidden (false otherwise)
- *className*: name of the CherryClass the method belongs to

Let's go back to our example: let's imagine that we want to use the regular header and footer for both pages *index* and *otherPage*, but we want a third page called *yetAnotherPage* that has its own header and footer. Here is what the code would look like:

```
CherryClass NormalHeaderAndFooter:
aspect:
    (method.type=='mask' and method.name!='yetAnotherPage') start:
        _page.append("<html><body>I'm the header<br><br>")
    (method.type=='mask' and method.name!='yetAnotherPage') end:
        _page.append("<br><br>I'm the footer</body></html>")

CherryClass Root(NormalHeaderAndFooter):
mask:
    def index(self):
            Hello, world !
    def otherPage(self):
            I love cherry pie !
    def yetAnotherPage(self):
        <html><body bgcolor=red>
            I love cherry pie !
        </body></html>
```

As we can see, we use the aspect condition to indicate which method the aspect will apply to. Note that the "method.type='mask'" condition is useless, since all methods of *Root* are masks. So it is only there to show you an example of an aspect condition.

11.3 How is it used in CookieAuthenticate and HttpAuthenticate

Now that you know how it works, you can look at the source code for these two modules and you should be able to understand it.

Basically, when you declare a CherryClass that inherits from CookieAuthenticate or HttpAuthenticate, some code is automatically added at the beginning of each of your masks and views. This code checks if the user is authenticated or not. If not, it returns the login page instead of the regular page ...

CHAPTER

TWELVE

How to create a spinning server and then debug it

12.1 Creating a spinning server

By "spinning" server, I mean a server that’s stuck in a loop while trying to build a page.

The easiest way to achieve that is:

```
CherryClass Root:
view:
    def index(self):
        while 1: pass
```

Compile that, start the server and try to request the home page ... :-) There you have it: a nice "spinning" server. Of course, the page will never be rendered, and if you try to load the same page from another browser, your browser won’t even be able to connect to the CherryPy server because it’s still busy trying to serve the first page (unless you have multiple server processes running).

The way you can tell that your server is spinning is very easy: you’re no longer able to connect to it, and the cpu time used by the process (which is displayed when you do a *ps* on Unix for instance) keeps growing, meaning that the process is not stalled, but it’s actually doing something.

You might be thinking "I’m not that stupid ! I will never write such an obvious loop !".

But here is another less trivial case that actually happened to me (and that’s the reason why I wrote this HowTo):

```
CherryClass Root:
function:
    def extractText(self, code):
        # Remove text between < and >
        while 1:
            i=code.find('<')
            if i==-1: break
            j=code.find('>', i)
            if j!=-1: code=code[:i]+' '+code[j+1:]
        return code
mask:
    def index(self, code=''):
        <html><body>
            Extracted text: <py-eval="self.extractText(code)">
            <form action="index">
            New text: <textarea name="code"></textarea>
            <input type=submit>
            </form>
        </body></html>
```

The *extractText* function takes some HTML code as an input and strips out the HTML tags. The output is the text that's extracted from the HTML code. The *index* mask just provides a textarea-based interface to test the function.

If you compile this code, run it and test it with some HTML code, it might run fine for a while, but in some cases it will start "spinning" ! The reason is that the *extractText* function is buggy in some cases: if the code is not proper HTML (for instance, a tag is opened with a "<" but never closed with a ">"), the function will enter an infinite loop. Correcting that is very easy once you know what's happening (just add "else: break" after the second "if" of the function).

But when this is happening on your production server, all you see is that your CherryPy server sometimes start spinning. Besides, the server might run fine for days, and start spinning all the sudden, making it very hard to reproduce it in your development environment and thus almost impossible to find out where it comes from !

The next section explain how to easily debug that to track down the culprit ...

12.2 Debugging a spinning server

The following method only works on Unix because it uses the *gdb* debugger.

All you have to do is wait for your CherryPy server to start spinning. Once it happens, fire up *gdb* using the name of the Python program that's running your CherryPy server (with the correct version). This might be for instance:

```
gdb python2.1
or
gdb python2.2
```

Once you're in *gdb*, just attach to the CherryPy process (you can get the process number with the *ps* command):

```
attach 7457
```

At this point, *gdb* will display a bunch of messages saying that it's reading and loading some symbols.

Now comes the clever trick: run the following command in gdb:

```
call PyRun_SimpleString("import sys, traceback; sys.stderr=open('/tmp/tb','w',0); traceback.pri
```

This will save the traceback in the '/tmp/tb' file. Just exit gdb and look at this file ... It should be obvious where the server was stuck. In our example, the file contained the following lines:

```
File "TestServer.py", line 454, in ?
  try: _serveForever(_masterSocket)
File "TestServer.py", line 215, in _serveForever
  _handleRequest(_wfile)
File "TestServer.py", line 363, in _handleRequest
  response.body=eval("%s.%s(%s)"%(_myClass,_function, _paramStr))
File "<string>", line 0, in ?
File "TestServer.py", line 61, in index
  _page.append(str(self.extractText(code)))
File "TestServer.py", line 55, in extractText
  if j!=-1: code=code[:i]+' '+code[j+1:]
File "<string>", line 1, in ?
```

PS: Thanks to Barry Warsaw for this trick (which was originally posted to a Zope mailing list).

CHAPTER

THIRTEEN

How to create an XML-RPC server with CherryPy

Writing an XML-RPC service with CherryPy is very easy. It works very much the same way as a regular web service. It is base on the *xmlrpclib* module. This module is included with Python-2.2 and higher, but if you use a lower version, you'll have to download it from http://www.pythonware.com.

All you have to do is add the keyword xmlrpc after the definition of the mask or view, and before the colon. Now, that view or mask will be available to XML-RPC requests.

13.1 Basic Example

Let's take an example. Just create a file called testXmlRpc.cpy containing the following code:

```
CherryClass Root:
view:
    def add(self, a, b, c) xmlrpc:
        # Return the sum of three numbers
        return a+b+c

CherryClass Xml_rpc:
view:
    def reverse(self, label) xmlrpc:
        # Reverse the characters of a string
        newStr=''
        for i in range(len(label)-1,-1,-1):
            newStr+=label[i]
        return newStr
```

Also, create a configuration file (textXmlRpcServer.cfg) to tell the server to accept XML-RPC requests:

```
[server]
typeOfRequests=xmlRpc,web
```

Compile the source file and start the server: you have an XML-RPC server running.

Now let's write a client. Just create a file called testXmlRpcClient.py containing the following code:

```
import xmlrpclib
testsvr = xmlrpclib.Server("http://localhost:8000")
print testsvr.add(1,2,3)
print testsvr.xml.rpc.reverse("I love cherry pie")
```

Run the client and you should see:

```
[remi@serveur]$ python xmlrpcTestClient.py
6
eip yrrehc evol I
[remi@serveur]$
```

13.2 Multiple Servers

CherryPy lets you define your XML-RPC server URIs any way you want. In our example client above, we used:

```
testsvr = xmlrpclib.Server("http://localhost:8000")
print testsvr.xml.rpc.reverse("I love cherry pie")
```

However, either of the following would work just as well.

```
testsvr = xmlrpclib.Server("http://localhost:8000/xml")
print testsvr.rpc.reverse("I love cherry pie")
```

or

```
testsvr = xmlrpclib.Server("http://localhost:8000/xml/rpc")
print testsvr.reverse("I love cherry pie")
```

Note that in these two examples, the *add* view in the *Root* CherryClass would not be available to the XML-RPC client. This gives you the flexibility to define your XML-RPC APIs to make different methods available at different URIs.

PS: Note that you can also declare a whole CherryClass as xmlrpc. This is equivalent to declaring each mask and view of this CherryClass as xmlrpc. For instance, the basic example of this HowTo could have been written as:

```
CherryClass Root xmlrpc:
view:
    def add(self, a, b, c):
        # Return the sum of three numbers
        return a+b+c

CherryClass Xml_rpc xmlrpc:
view:
    def reverse(self, label):
        # Reverse the characters of a string
        newStr=''
        for i in range(len(label)-1,-1,-1):
            newStr+=label[i]
        return newStr
```

CHAPTER

FOURTEEN

How to make hidden masks or views

14.1 Introduction

Masks and views usually correspond to URLs. For instance, if one writes the following code:

```
CherryClass CommonMasks:
mask:
    def redLabel(self, label):
        <b><font color=red py-eval="label"></font></b>
CherryClass Root:
mask:
    def index(self):
        <html><body>
            Hello, <py-eval="commonMasks.redLabel('world')">
        </body></html>
```

The URL http://localhost:8000 will correspond to *root.index*.

That's fine, but this also means that if someone artificially types the URL http://localhost:8000/commonMasks/redLabel?label=IHateCherryPy, they will get the result of the *redLabel* mask.

In this case, it's not a very big deal because the *redLabel* mask doesn't do anything important, but in some cases this might be a problem. That's why **hidden** masks and views were included in CherryPy-0.8.

14.2 How it works

All you have to do is add the keyword **hidden** after the definition of the mask or view, and before the colon. In our example, one could write:

```
CherryClass CommonMasks:
mask:
    def redLabel(self, label) hidden:
        <b><font color=red py-eval="label"></font></b>
CherryClass Root:
mask:
    def index(self):
        <html><body>
            Hello, <py-eval="commonMasks.redLabel('world')">
        </body></html>
```

All this means is that the *redLabel* mask can no longer be accessed directly from the browser. But it can be called from another mask or view.

It is also possible to declare that an entire CherryClass is hidden, like this:

```
CherryClass CommonMasks hidden:
mask:
    def redLabel(self, label):
        <b><font color=red py-eval="label"></font></b>
CherryClass Root:
mask:
    def index(self):
        <html><body>
            Hello, <py-eval="commonMasks.redLabel('world')">
        </body></html>
```

In this case, all masks and views of *CommonMasks* will be hidden.

CHAPTER

FIFTEEN

How to control logs

By default, a CherryPy server outputs all logs to the console. But you can control that by using two configuration variables in the **[server]** section of the configuration file:

- **logToScreen**: This indicates if the logs should be sent to the console or not. Possible values are 0 or 1, default value is 1. Example:

```
[server]
socketPort=80
logToScreen=0
```

- **logFile**: Here you can specify the path of a file where all logs will be stored. Example:

```
[server]
socketPort=80
logToScreen=0
logFile=/mydir/myLog.log
```

If these options are not enough for you (for instance, you might be running CherryPy on some embedded device with weird logging constraints), you can define your own special function called *logMessage*, which takes two arguments (including the optional level). The default behaviour of this function is to output the message to screen and/or to a file according to the *logToScreen* and *logFile* configuration variables.

The following code is an example of how to write your own *logMessage* function:

```
def logMessage(message, level=0):
    # Only print message if level < 5
    if level < 5:
        print "Here is your message:", message

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

Note that the first line **def logMessage(message, level=0):** has to be exactly that !

CHAPTER

SIXTEEN

How to use sessions

16.1 Introduction to sessions

The main difference between a web application and a regular application is that in a web application, each page requested by a client is independant from the other pages requested by the same client. In other words, we have to start from scratch for each page.

Of course, for any serious application, this is not acceptable and we need to keep some data about a given user across several pages.

Let's assume for example that we want to keep the first name and the last name of the user across several pages. We ask the user for his first name and last name on the first page of the website (using a form) and we need to use that data in other pages of the site.

First of all, we have basically two options:

- keep the first name and the last name from one page to the next on the client side
- store the first name and the last name somewhere on the server side, and only keep a pointer to that data on the client side: **this is what sessions do**

If we choose the first option, we have several ways to do this:

- Keep the first name and the first name in the URL: each time there is a link to another page, we could add the following arguments to the URL: "firstName=....&lastName=..."
- Keep the data in hidden fields of a form: in each page, we could have a form with two hidden fields like this:

```
<form method="post" name="myForm" action="dummy">
    <hidden name="firstName" value="...">
    <hidden name="lastName" value="...">
</form>
```

  Everytime there is a link to another page, we have to submit the form to get the data. We can use something like this:

```
<a href="#"
   onClick="
      document.myForm.action='...put link here...';
      document.myForm.submit();
      return false;">...</a>
```

- Store the first name and the last name in a cookie

The first two options are not really handy as they force you to use extra code for each link. The third option is better but it is not very handy if we have lots of data to keep about a user.

So another option is to store the data on the server side and to just keep a pointer to that data on the client side. This is what sessions do ...

16.2 Possible implementations for sessions

The pointer to the data is usually called a **sessionId**. Since we need to keep the sessionId from one page to the next on the client side, we still have the same options as described in the previous section:

- Store the sessionId in the URL (this is why you sometimes see long URLs for some sites like http://domain/page?sessionId=dsqlkjsqlkdsklsjsk7987987987987dqkshsqjkhsq798798)
- Store the sessionId in the hidden field of a form future.
- Store the sessionId in a cookie

The session data itself (in our example: the first name and the last name) is stored on the server side. Again, there are many options to store it:

- Store it in RAM
  - Advantage: very fast; can store any python object
  - Disavantage: doesn't work in in a multi-process environment; we lose the data when the server is stopped
- Store it in the filesystem
  - Advantage: never lose data; works in a multi-process environment
  - Disavantage: slow; lots of reads/writes to disk; can only store pickable python objects
- Store it in a cookie in the client's browser
  - Advantage: works in multi-thread or multi-process environments. Server is not vulnerable to attacks that consist in artificially starting thousands of sessions on the server to bring it to its knees.
  - Disavantage: can only store pickable python objects. Session data **must** be quite small (some browsers limit the size of cookie data to 4KB)
- Store it in a database
  - Advantage: never lose data; works in a multi-process environment
  - Disavantage: slower than RAM; lots of reads/writes to database; can only store pickable python objects
- Store it anywhere you can think of ...

Some systems can also mix some of these options: for instance, session data can be stored in RAM and saved to disk or to a database once in a while.

Another option is to store everything in RAM and save it to disk when the server shutsdown (although this might be dangerous because we might not have time to save it if the server crashes or is killed badly).

Also, session data might not change too often so another option is to store it in RAM and save it to disk or to a database only when it changes.

16.3 Sessions implementation in CherryPy

The following options are supported in CherryPy:

- sessionId: for now, the sessionId is always stored in a cookie
- session data: three options are build in and you can implement very easily any storage method you want ...
  - Store everything in RAM; never save it to disk or to a database
  - Store everything in the filesystem; read and save the data for each request
  - Store the session data in the client browser, as a cookie
  - Write your own storage functions so you can store the data wherever you want

You should use the first option (RAM) if you don't use multiple processes (using multiple threads is fine) and if you don't care about losing session data when the server is stopped/restarted.

You should use the second option (filesystem) if you don't have a database and don't want to lose data.

The third option is quite interesting but only works if the session data for each user is quite small (some browsers limit the size of cookies to 4KB).

The fourth option (storing your session data in a database) is the recommended option for most "real-world" websites.

16.4 Configuration variables used to control sessions

In order to use sessions, you must first enable sessions by setting a few configuration variables on the config file, under the *[session]* scope. The new configuration options are:

- *storageType*: Can be either "ram", "file", "cookie" or "custom": this tells whether you want to store session data in RAM, to disk, in a cookie, or write your own storage functions.
- *storageFileDir*: This must be set if you set *storageType* to "file". Set it to the directory where the session data will be stored
- *timeout*: Number of minutes after which a session expires if there was no activity. The default is 60 minutes.
- *cleanUpDelay*: If cleanUpDelay is set to N minutes, then the CherryPy server will check every N minutes if there are old/expired sessions that need to be cleaned up. The default is 60 minutes.
- *cookieName*: Name of the cookie that stores the sessionId: The default is "CherryPySession"

16.5 Cleaning up old sessions

If *storageType* is either "ram", "file" or "cookie", then CherryPy will automatically clean up expired sessions for you. If *storageType* is set to "custom", then you have to write your own code to clean up old sessions in a special function called *cleanUpOldSessions*.

16.6 Using sessions in your code

Once you have enabled sessions in the config file, the way it works is very easy: CherryPy just makes available for you a dictionary that can be accessed through *request.sessionMap*. You can use this dictionary to store the data that you want to keep about a perticular client.

The reason why *sessionMap* is a member variable of *request* is because this makes it thread-safe.

16.7 Example

For instance, if you want to store session data in RAM, if you want sessions to expire after 2 hours, and if you want CherryPy to clean up expired sessions every hour, use the following config file ('RootServer.cfg'):

```
[session]
storageType=ram
timeout=120
cleanUpDelay=60
```

If you want to store session data to disk (in a directory called '/home/user/sessionData'), use the following config file:

```
[session]
storageType=file
storageFileDir=/home/user/sessionData
```

The following example is a trivial page view counter:

```
CherryClass Root:
mask:
    def index(self):
        <py-code="
            count=sessionMap.get('pageViews', 0)+1
            sessionMap['pageViews']=count
        ">
        <html><body>
            Hello, you've been here <py-eval="count"> time(s)
        </body></html>
```

16.8 Storing session data in a database (or anywhere else)

From CherryPy-0.9-gamma and later, you can now write your own custom functions to save/load session data. This allows you to store them wherever you want. In order to do so, all you have to do is declare the three following special functions somewhere in your code:

- *saveSessionData(sessionId, sessionData, expirationTime)*: Store the sessionData and expiration time for this sessionId somewhere. *expirationTime* is a *float* as returned by the *time.time()* function.
- *loadSessionData(sessionId)*: Retrieve the sessionData and expiration time for this sessionId and returns them as a tuple
- *cleanUpOldSessions()*: Delete expired sessions

The following example shows how to store the session data in a MySql database. It assumes that there is a table called "session_data":

```
mysql> create table session_data(sessionId varchar(50), sessionData text, expirationTime int un
Query OK, 0 rows affected (0.00 sec)

mysql> describe session_data;
+----------------+------------------+------+-----+---------+-------+
| Field          | Type             | Null | Key | Default | Extra |
+----------------+------------------+------+-----+---------+-------+
| sessionId      | varchar(50)      | YES  |     | NULL    |       |
| sessionData    | text             | YES  |     | NULL    |       |
| expirationTime | int(10) unsigned | YES  |     | NULL    |       |
+----------------+------------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```

Note that we've chosen to use an *int* column to store the expiration time, which is easier because that's how this variable is passed to the *saveSessionData* function. We could have chosen to use a *datetime* column but then we would have to convert *expiration* time into a suitable format to insert it into the table (and convert it back in *loadSessionData*).

Here is an example code that uses the *session_data* MySql table to store session data:

```
use MySql
import pickle, base64, StringIO, time
CherryClass MyDb(MySql):
function:
    def __init__(self):
        self.openConnection('localhost', 'test', '', 'test')

def saveSessionData(sessionId, sessionData, expirationTime):
    # Pickle sessionData and base64 encode it
    f = StringIO.StringIO()
    pickle.dump(sessionData, f, 1)
    dumpStr = f.getvalue()
    f.close()
    dumpStr = base64.encodestring(dumpStr)
    myDb.query("delete from session_data where sessionId='%s'" % sessionId)
    myDb.query("insert into session_data values('%s', '%s', %s)" % (
        sessionId, dumpStr, int(expirationTime)))

def loadSessionData(sessionId):
    sessionList = myDb.query("select sessionData, expirationTime from session_data where sessio
    if not sessionList:
        return None
    dumpStr = base64.decodestring(sessionList[0][0])
    f = StringIO.StringIO(dumpStr)
    sessionData = pickle.load(f)
    f.close()
    return (sessionData, sessionList[0][1])

def cleanUpOldSessions():
    now = time.time()
    myDb.query("delete from session_data where expirationTime < %s" % int(now))

CherryClass Root:
view:
    def index(self):
        i = request.sessionMap.get('counter', 0) + 1
        request.sessionMap['counter'] = i
        return str(i)
```

CHAPTER

SEVENTEEN

How to use Psyco with CherryPy

Psyco is a python module that can improve the speed of some python programs. Psyco can be found here: http://psyco.sourceforge.net.

Pysco has little impact on the speed of the CherryPy code itself (the code is already really fast :-) so you should only use it if you think it can improve the speed of the code that **you** wrote for your site (for instance, if you use some number-crunching functions to build your pages).

Also, some users reported that Psyco introduces big memory leaks in CherryPy web apps. So if you try to use it and notice that your site leaks memory, you'll know where to look ...

In order to enable Psyco with CherryPy, you should have Psyco installed on your machine and then all you have to do is to use the "initProgram" special function and to put the following code in it:

```
def initProgram():
    import psyco
    psyco.full()
```

CHAPTER

EIGHTEEN

How to use cookies with CherryPy

CherryPy uses the *Cookie* module from python and in particular the *SimpleCookie* object type to handle cookies. More information can be found here: http://www.python.org/doc/current/lib/module-Cookie.html

18.1 Setting cookies

In order to send a cookie to a browser, you have to use the global variable *response.simpleCookie*, which is a *SimpleCookie* object.

The following code shows how to set a cookie in your CherryPy code:

```
CherryClass Root:
view:
    def index(self):
        response.simpleCookie['cookieName']='cookieValue'
        response.simpleCookie['cookieName']['path']='/'
        response.simpleCookie['cookieName']['max-age']=3600
        response.simpleCookie['cookieName']['version']=1
        return "<html><body>Hello, I just sent you a cookie</body></html>"
```

18.2 Reading cookies

Cookies that are sent by a browser are stored in the global variable *response.simpleCookie*, which is a *SimpleCookie* object.

The following code shows how to read a cookie in your CherryPy code:

```
CherryClass Root:
mask:
    def index(self):
        <html><body>
            Hi, you sent me <py-eval="len(request.simpleCookie)"> cookies.<br>
            Here is a list of cookie names/values:<br>
            <py-for="cookieName in request.simpleCookie.keys()">
                <py-eval="cookieName+': '+request.simpleCookie[cookieName].value"><br>
            </py-for>
        </body></html>
```

CHAPTER

NINETEEN

How to use Cheetah templates with CherryPy

Cheetah is a templating language for python. CherryPy comes with its own templating language (CGTL and CHTL), but if you prefer to use Cheetah, that's very easy to do. More about Cheetah can be found at: http://www.cheetahtemplate.org/.

Here is a sample code that shows how to do it:

```
##################
# File Root.cpy
##################
from Cheetah.Template import Template
CherryClass Root:
view:
    def index(self):
        template = Template(file='index.tmpl')
        template.colors = ['AA', 'BB', 'CC']
        return template.respond()

##################
# File index.tmpl
##################
<html>
<head>
<title>Cheetah Experiment</title>
</head>
#attr $colors = []
#if $colors is []
    #set codelist = ['00','33','66','99','CC','FF']
#else
    #set codelist = $colors
#end if
<body>
<table border=1>
#for $r in $codelist
    #for $g in $codelist
        <tr>
        #for $b in codelist
        #set color = '#%s%s%s'%(r,g,b)
        <td bgColor = $color>   $color   </td>
        #end for
    #end for
#end for
</table>
</body>
</html>
```

Note that this example is not optimized because the template file will be read/parsed/rendered each time the page is displayed. It is quite easy to change that code so it only does it once, or everytime the template file is changed.

Many thanks to "ToddB" for contributing this code :-)

CHAPTER

TWENTY

How to use streaming with CherryPy

If you need to return a very big page to a browser, then you might want to use streaming, which means that CherryPy will start sending the page back to the browser before the page is completely built, and the page will be returned in chunks.

All you need to do to use streaming is to use the *response.wfile* variable, which contains the socket used to send the response back to the browser. You need to write **all** the data on the socket (including the response header).

You also need to tell CherryPy that you've already sent the response to the browser and therefore it doesn't have to do it. To do so, you need to set the *response.sendResponse* variable to 0.

The following code shows an example of how to use streaming:

```
import time
CherryClass Root:
view:
    def index(self):
        response.wfile.write("HTTP/1.1 200\r\n")
        response.wfile.write("Content-Type: text/plain\r\n\r\n")
        response.wfile.write("First line. Sleeping 2 seconds ...\n")
        time.sleep(2)
        response.wfile.write("Second line. Sleeping 2 seconds ...\n")
        time.sleep(2)
        response.wfile.write("Third and last line")
        response.sendResponse = 0
        return "" # The view still needs to return a string
```

CHAPTER

TWENTYONE

Sample deployment configuration for a real-world website

This HowTo describes a sample deployment configuration that I believe is well adapted for most "real-world" websites. It is the configuration used by the http://www.cherrypy.org website itself.

21.1 Hardware

The website runs on the Celeron 1.3Ghz with 512MB RAM and shares the machine with about 30 other sites. The choice of the hardware depends mostly on your budget and how much traffic you plan to get...

21.2 Software environment

The site runs on Linux RedHat, but I believe it would run just as well on other OSes (including Windows). It runs on Python-2.3, but I believe it would run just as well on Python-2.2 or Python-2.1. However, Python-2.3 seems to be a bit faster than older versions.

21.3 CherryPy version

The site usually uses the latest CVS version from CherryPy (currently the 0.9-final) version. The 0.9 version seems to be much more stable than previous versions so if you are running an older version, I highly recommend that you upgrade to that version.

21.4 CherryPy server configuration

The site is deployed as a thread-pool server. I believe this is the best option for most websites. It is configured to run with 10 threads and since each page is really fast to build, the site should be able to support many (much more than 10) concurrent users...

The number of threads that you should use for your site depends on many criteria, including the number of concurrent users you plan to have, the average time your pages take to build and the power (CPU and RAM) of the machine that will run the site.

21.5 CherryPy server deployment

The site is running "exposed" on port 80, which means that there is no other webserver (like Apache) involved.

In order to listen on port 80, the server has to be started by the user "root". However, running the server as "root" is not very safe, so we use the special function *initAfterBind* to switch the user running the server to another user. The code is very simple:

```
def initAfterBind():
    import os
    # We must switch the group first
    os.setegid(500)  # Replace with desired user id.
    os.seteuid(2524) # Idem
```

Also, since the server runs a pool of threads, we must switch the user for all these threads as well (otherwise, only the parent thread will be switched). To do so, we just use the *initThread* special function, like this:

```
def initThread(threadIndex):
    import os
    # We must switch the group first
    os.setegid(500)  # Replace with desired user id.
    os.seteuid(2524) # Idem
```

21.6 Database configuration

The site uses MySql as a database backend to log the requests that are being made to the server. Therefore, each request triggers a database write.

Since the MySQLdb driver doesn't work well (from my experience) if we have several threads sharing the same database connection, each thread is given its own database connection, so it doesn't interfer with other threads.

This is done very easily using the *initThread* special function and the *request* special variable, like this:

```
def initThread(threadIndex):
    time.sleep(threadIndex * 0.2) # Sleep 0.2 seconds between each new database connection to m
    request.connection = MySQLdb.connect('host', 'user', 'password', 'name')
```

This takes advantage of the fact that *request* is a special thread-aware class instance (so each thread can safely set/get member variables without have to worry about other threads)

Then, when we need to run a query, we just use code like this:

```
c=request.connection.cursor()
c.execute(query)
res=c.fetchall()
c.close()
```

And also, the forum on the website runs on top of ZODB and since ZODB doesn't support multiple threads by itself, I had to install ZEO so each thread also behaves as a ZEO client.

21.7 Sessions

The site also uses sessions (mostly for the forum, and also for one page in the online demo). Sessions are configured to store the data in RAM. Since the code for managing sessions is thread-safe, all threads have access to the same session data and everybody is happy :-)

I believe that other storage types for sessions (file, database, cookie) would work just as well.

21.8 Results

As I write these lines, the site has been running with no problem with this new configuration for about 20 days and I never had to restart it, proving that it is quite stable. Also, the RAM used by the threads has never increased, proving that there is no memory leaks ... It is true that the CherryPy.org website is not a "high-traffic" website, but it is still a good sign.

Note that if you're afraid that your CherryPy server might crash while you're sleeping, you can always set up a cronjob to check that the site is still running and to restart it if it isn't ...

CHAPTER

TWENTYTWO

How to stream uploaded files directly to disk

By default, CherryPy handles in the same way all data posted to the server through a form. In all cases, CherryPy converts that data to a string or a list of strings. This data can be a short string or a big file that's being uploaded.

In most cases, this is very convenient and it works very well. However, if your application requires users to upload very big files, then converting them to a string and having this string in memory can be a problem ...

In that case, you may want to write these files directly to a file instead of having them in memory.

Here is how this can be done:

- Use the *initRequestBeforeParse* special function to tell CherryPy not to parse the POST data itself (by setting *request.parsePostData* to 0).
- Use the *FieldStorage* class of the *cgi* module to parse the POST data (by reading *request.rfile*) and stream the uploaded file to disk.

Here is an example code that does this:

```
import cgi

def initRequestBeforeParse():
    if request.path == 'postFile':
        request.parsePostData = 0

CherryClass Root:
mask:
    def index(self):
        <html><body>
        <form method=post action=postFile enctype="multipart/form-data">
            Upload a file: <input type=file name=myFile><br>
            <input type=submit>
        </form>
        </body></html>

view:
    def postFile(self):
        # Use cgi.FieldStorage to parse the POST data
        dataDict = cgi.FieldStorage(fp=request.rfile, headers=request.headerMap, environ={'REQU

        value = dataDict['myFile']

        # Value has 2 attributes:
        #    - filename contains the name of the uploaded file
        #    - file is an input stream opened for reading

        f = open('/tmp/myFile', 'wb')
        while 1:
            data = value.file.read(1024 * 8) # Read blocks of 8KB at a time
            if not data: break
            f.write(data)
        f.close()

        return "<html><body>The file has been saved in /tmp/myFile</body></html>"
```

Note that the file is not streamed directly from the browser to '/tmp/myFile'. Instead, it is saved in a *tempfile* by the *cgi* module and then streamed from this *tempfile* to '/tmp/myFile'.