



CHERRYPY

ARIHARASUDHAN

aravindariharan@gmail.com



CherryPy

It is a Pythonic Object Oriented Web Framework. It means, We can build web applications in the same way we build any Object-Oriented Python Program. It leads to reduced lines of code and development time. It is used in many sites. CherryPy is a Python library providing a friendly interface to the HTTP protocol for Python developers. Remi Delon created the CherryPy. It is close to the MVC pattern. CherryPy would map a URL and its query string into a Python method call, for example: `http://myhost.net/eat?food=cherry` would map to `eat(food='cherry')`.



CherryPy

*A minimalist
Python Web
Framework*



aravindariharan@gmail.com



The Jargons

Web Server : A web server is the interface dealing with the HTTP protocol. Its goal is to transform incoming HTTP requests into entities that are then passed to the application server and also transform information from the application server back into HTTP responses.

Application : An application is a piece of software that takes a unit of information, applies business logic to it, and returns a processed unit of information.

Application Server : An application server is the component hosting one or more applications.

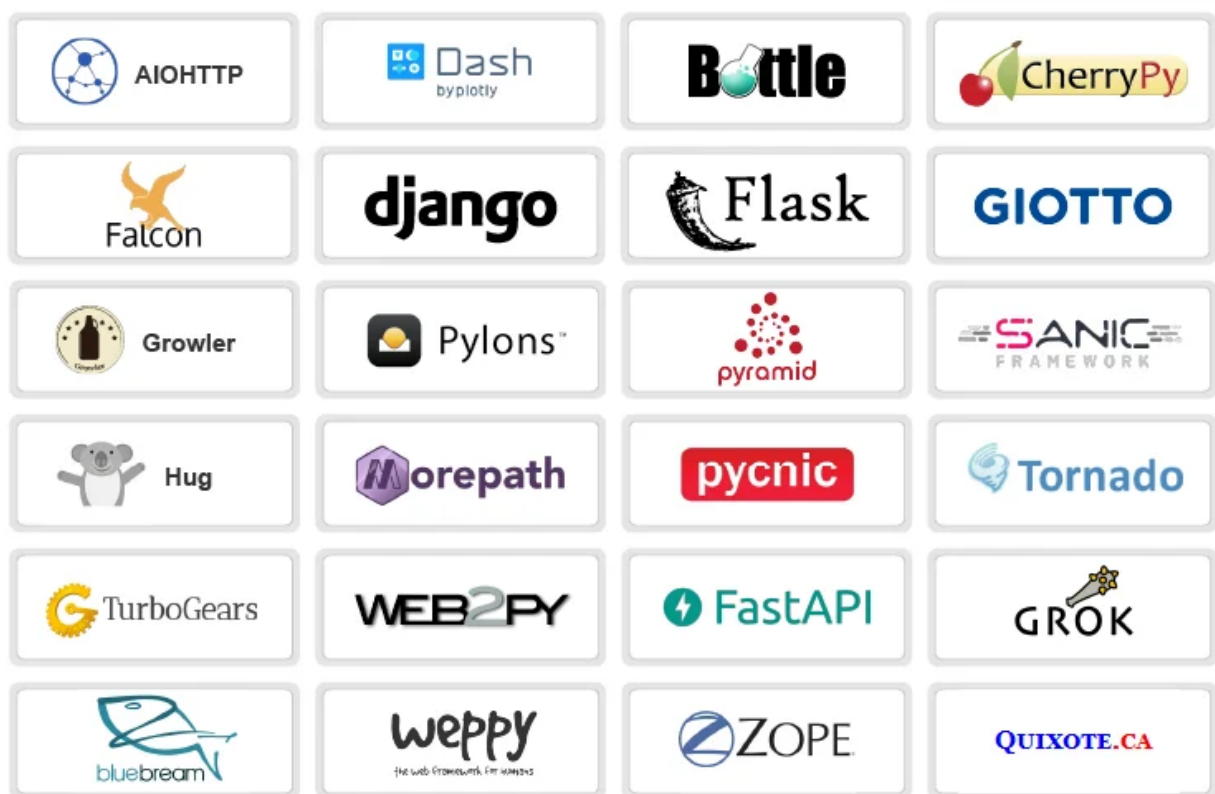
Web Application Server : A web application server is simply the aggregation of a web server and an application server into a single component.

CherryPy is a Web Application Server..



WebServer&Framework

A web server responds to HTTP requests, sending back the data the request asked for. A web framework provides you the skeleton of an application, including pre-built code to listen at a port, parse HTTP requests, and format HTTP responses. You use the skeleton to do the boilerplate work and add your own code to that skeleton to customize it. Some Web Frameworks in Python are,



aravindariharan@gmail.com



Why CherryPy ?

Self Contained : CherryPy don't require any 3rd party Python packages to work. It is simple.

Not Intrusive : CherryPy stays as much as possible out of the way of its users.

Multithreaded Web Server : CherryPy application embeds its own multithreaded web server. It supports various web servers.

Built-In HTTP Server

CherryPy comes with its own web (HTTP) server. The goal of this decision was to make CherryPy self-contained and allow users to run a CherryPy application within minutes of getting the library. As the name implies, the web server is the gateway to a CherryPy application through which all HTTP requests and responses have to go. To start the web server : `cherrypy.server.quickstart()`



CherryPy is not..

CherryPy is among the oldest web framework available for Python, yet many people aren't aware of its existence. Because, CherryPy is not a complete stack with built-in support for a multi-tier architecture. It doesn't provide frontend utilities nor will it tell you how to speak with your storage. Instead, CherryPy is used to let the developer make those decisions. Yet, it is used in Netflix, CherryMusic, Linstic (Sticky Notes), TurboGears and so on.

Some Requirements..

doc, json, ssl, xcgi, memcached_session, routes_dispatcher and testing are some of the requirements of the features in CherryPy. CherryPy supports Python 3.6 through to 3.11



First App

```
1 import cherrypy
2 class MyApp:
3     @cherrypy.expose
4     def index(self):
5         return 'Hello Cherries'
6
7 cherrypy.quickstart(MyApp())
8
```

@cherrypy.expose makes the method accessible by website. Once we run this script, we could see,

```
In [1]: runfile('/home/ari-pt7127/CherryPy/first.py', wdir='/home/ari-pt7127/CherryPy')
[09/Feb/2023:10:35:09] ENGINE Listening for SIGTERM.
[09/Feb/2023:10:35:09] ENGINE Listening for SIGHUP.
[09/Feb/2023:10:35:09] ENGINE Listening for SIGUSR1.
[09/Feb/2023:10:35:09] ENGINE Bus STARTING
CherryPy Checker:
The Application mounted at '' has an empty config.

[09/Feb/2023:10:35:09] ENGINE Started monitor thread 'Autoreloader'.
[09/Feb/2023:10:35:09] ENGINE Serving on http://127.0.0.1:8080
[09/Feb/2023:10:35:09] ENGINE Bus STARTED
```

We can go <http://127.0.0.1:8080> to see the output.



Hello Cherries

aravindariharan@gmail.com



Different URLs

```
1 import cherrypy
2 class MyApp:
3     @cherrypy.expose
4     def index(self):
5         return 'Hello Cherries'
6
7     @cherrypy.expose
8     def myname(self):
9         return 'Ariharasudhan'
10
11 cherrypy.quickstart(MyApp())
```

Different URL's will take us to different applications. If we look here, we have an index method, which will be executed when we go to <http://127.0.0.1:8080> by default. Meanwhile, myname() method is executed only when you go to the address <http://127.0.0.1:8080/myname>.



Hello Cherries



Ariharasudhan

aravindariharan@gmail.com



URLs with parameters

We can pass parameter values in URLs. Consider the following `myage()` with the default age of 20. This age value can be altered with <http://127.0.0.1:8080/myname?age=21>

```
1 import cherrypy
2 class MyApp:
3     @cherrypy.expose
4     def myage(self, age=20):
5         return "You are "+str(age)+" years old!"
6
7 cherrypy.quickstart(MyApp())
8
```

← → ↻ 127.0.0.1:8080/myage
You are 20 years old!

← → ↻ 127.0.0.1:8080/myage?age=21
You are 21 years old!



Talking to the server

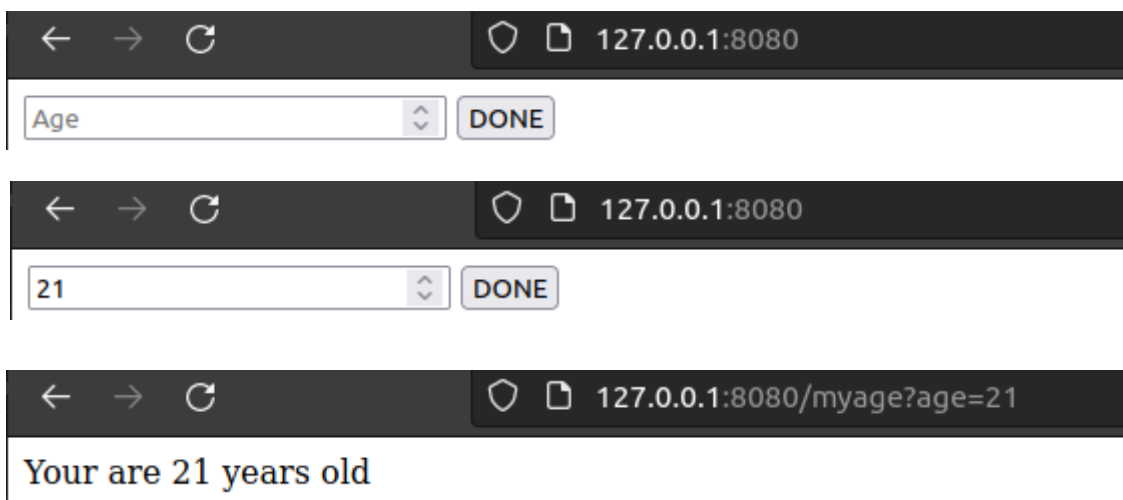
We can simply return a form which gets the value from us and takes it to the `myage()` method which will return the age.

```
import cherrypy

class MyApp:
    @cherrypy.expose
    def index(self):
        return """<html>
            <head></head>
            <body>
                <form method="get" action="myage">
                    <input type="number" placeholder="Age" name="age" />
                    <button type="submit">DONE</button>
                </form>
            </body>
        </html>"""

    @cherrypy.expose
    def myage(self, age=20):
        return "Your are "+str(age)+" years old"

cherrypy.quickstart(MyApp())
```



The first screenshot shows the initial state of the web application. The browser's address bar displays `127.0.0.1:8080`. The page contains a form with a text input field labeled "Age" and a "DONE" button.

The second screenshot shows the form after the user has entered the value "21". The "DONE" button remains visible.

The third screenshot shows the result of clicking the "DONE" button. The browser's address bar now shows `127.0.0.1:8080/myage?age=21`, and the page content has changed to "Your are 21 years old".



Session

It is normal to follow the user activity for a while. Let's enable sessioning in CherryPy using the configuration dictionary.

```
import cherrypy
class MyApp(object):
    @cherrypy.expose
    def index(self):
        return """<html>
            <head></head>
            <body>
                <form method="get" action="myname">
                    <input type="text" name="name" />
                    <button type="submit">DONE</button>
                </form>
            </body>
        </html>"""

    @cherrypy.expose
    def myname(self, name='Ari'):
        cherrypy.session['username'] = name
        uname = "Your name is "+name
        return uname

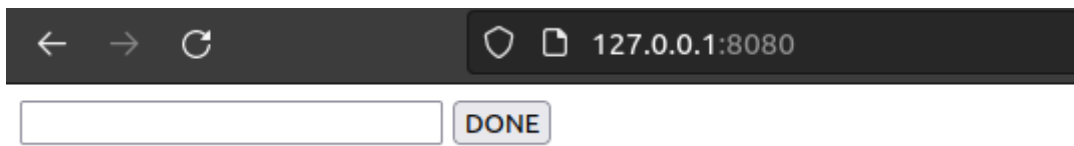
    @cherrypy.expose
    def display(self):
        return cherrypy.session['username']

conf = { '/': { 'tools.sessions.on': True } }
cherrypy.quickstart(MyApp(), '/', conf)
```

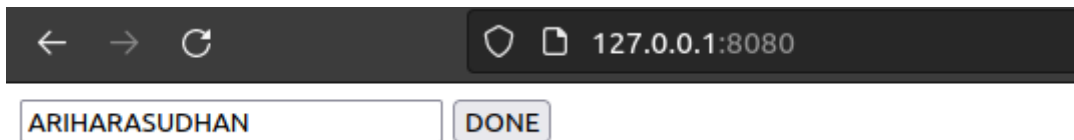
display() returns the session-trapped-value



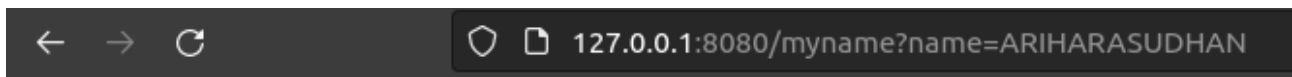
Go to the address..



Enter your name and click DONE..

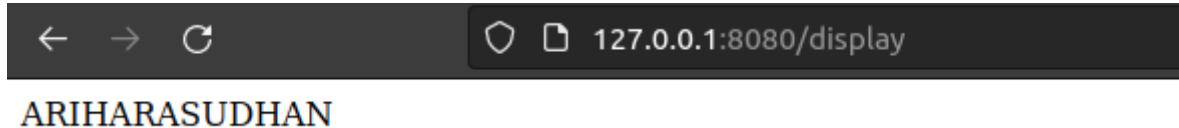


You are in..



Your name is ARIHARASUDHAN

Now, go to display..



Yeah.. Your name is trapped. By default, CherryPy will save sessions in the process's memory. It supports more persistent backends as well.



SESSION STORAGES

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

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

Memcached is a popular key-store on top of your RAM, it is distributed and a good choice if you want to share sessions outside of the process running CherryPy. Requires that the Python memcached package is installed, `cherrypy[memcached_session]`.

which may be indicated by installing
[/]

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

DEMERITS : Handling Many Users ,
Serializing



Static Files

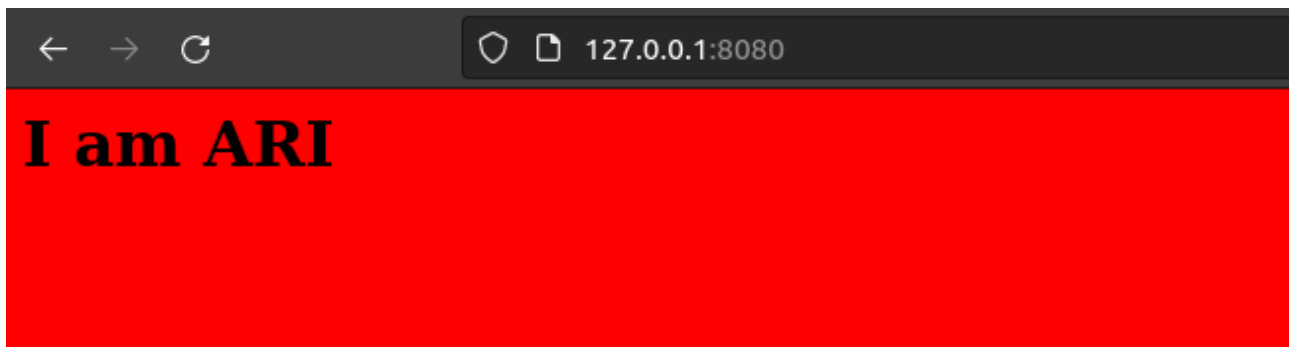
Web applications are usually also made of static contents such as javascript, CSS or images. CherryPy provides support to serve static content to end-users. First, we indicate the root directory of all of our static content. Then we indicate that all URLs which path segment starts with /static will be served as static content. We map that URL to the public directory.

```
import cherrypy
import os
class MyApp(object):
    @cherrypy.expose
    def index(self):
        return """<html><head>
<link href="/static/css/styles.css" rel="stylesheet">
</head>
<h1>I am ARI</h1>"""

conf = {
    '/': {
        'tools.staticdir.root': os.path.abspath(os.getcwd())
    },
    '/static': {
        'tools.staticdir.on': True,
        'tools.staticdir.dir': './public'
    }
}

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





We can add any sort of static contents such as images, JavaScript files in this way.

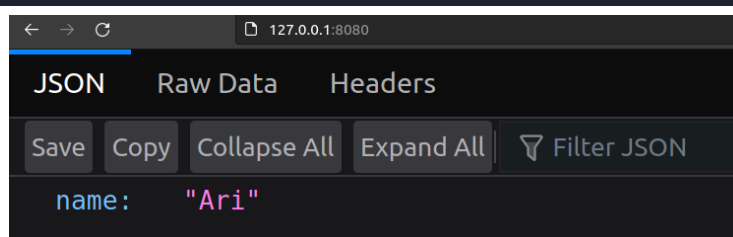
Returning JSON

Well! We love working with JSON. Let's add a functionality to the index method to return the JSON file by using a cherrypy tool. (json_in() is also there)

```
import cherrypy

class MyApp:
    @cherrypy.expose
    @cherrypy.tools.json_out()
    def index(self):
        return {'name': 'Ari'}

cherrypy.quickstart(MyApp())
```



aravindariharan@gmail.com



Ways of Exposing:

A Method can be made a handler by exposing that. There are two ways for it.

```
class Test()
    @cherrypy.expose
    def foo(self):
        return "Foo"

    def bar(self):
        return "Bar"
    bar.exposed=True
```

Here , both the foo() and the bar() methods will be exposed. i.e., these methods are now accessible in the web. If we need a method just to perform an additional task, we don't hafta expose it. For an instance, if you want to perform addition in the Integral Operation, we don't need to expose the add() method. Just expose the integrate() method which will get the addition result by calling the unexposed add() method.



CRUD On Session

```
import cherrypy

@cherrypy.expose
class NameWebService(object):

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

    def POST(self):
        some_string = "ARIHARASUDHAN"
        cherrypy.session['mystring'] = some_string
        return some_string

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

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

conf = {
    '/': {
        'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
        'tools.sessions.on': True,
        'tools.response_headers.on': True,
        'tools.response_headers.headers': [('Content-Type', 'text/plain')],
    }
}
cherrypy.quickstart(NameWebService(), '/', conf)
```

CRUD refers to the four basic operations a software application should be able to perform – Create, Read, Update, and Delete. Run this code and open a new terminal. We can go with the requests module for doing CRUD Operations over session dictionary.



```
In [12]: import requests

In [13]: s = requests.Session()

In [14]: r = s.get('http://127.0.0.1:8080/')

In [15]: r.status_code
Out[15]: 500

In [16]: r = s.post('http://127.0.0.1:8080/')

In [17]: r.status_code
Out[17]: 200

In [18]: r.text
Out[18]: 'ARIHARASUDHAN'

In [19]: r = s.put('http://127.0.0.1:8080/',{'another_string': 'ARI'})

In [20]: r.status_code
Out[20]: 200
```

```
In [22]: r = s.get('http://127.0.0.1:8080/')

In [23]: r.text
Out[23]: 'ARI'

In [24]: r = s.delete('http://127.0.0.1:8080/')

In [27]: r = s.get('http://127.0.0.1:8080/')

In [28]: r.status_code
Out[28]: 500
```

The HTTP 200 OK success status response code indicates that the request has succeeded. The HTTP 500 Internal Server Error server error response code indicates that the server encountered an unexpected condition that prevented it from fulfilling the request.



CRUD Operations on DB

Let's go with the sqlite database and have some relational stuffs. First of all, import the following modules. We're about to generate some random strings and store them in sqlite database.

```
import random
import sqlite3
import string
import time
import cherrypy
DB_STRING = "my.db"
```

Then, create a class for performing CRUD Operations.

```
@cherrypy.expose
class NameService(object):
    @cherrypy.tools.accept(media='text/plain')
    def GET(self):
        with sqlite3.connect(DB_STRING) as c:
            r = c.execute("SELECT * FROM ranstrs")
            return r.fetchone()

    def POST(self):
        some_string = ''.join(random.sample(string.hexdigits,8))
        with sqlite3.connect(DB_STRING) as c:
            c.execute("INSERT INTO ranstrs VALUES (?)",[some_string])
            return "INSERTED A RANDOM STRING"

    def PUT(self, another_string):
        with sqlite3.connect(DB_STRING) as c:
            c.execute("UPDATE ranstrs SET value=?", [another_string])
            return "UPDATED ALL ROWS"

    def DELETE(self):
        with sqlite3.connect(DB_STRING) as c:
            c.execute("DELETE * FROM ranstrs")
            return "DELETED ALL ROWS"
```

aravindariharan@gmail.com



We need to create a table in the database and drop it before and after starting and ending respectively.

```
def setup_database():
    with sqlite3.connect(DB_STRING) as con:
        con.execute("CREATE TABLE ranstrs (value)")

def cleanup_database():
    with sqlite3.connect(DB_STRING) as con:
        con.execute("DROP TABLE ranstrs")

conf = {
    '/': {
        'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
        'tools.response_headers.on': True,
        'tools.response_headers.headers': [('Content-Type', 'text/plain')],
    }
}

cherrypy.engine.subscribe('start', setup_database)
cherrypy.engine.subscribe('stop', cleanup_database)
cherrypy.quickstart(NameService(), '/', conf)
```

Now, use requests module for CRUD.

```
In [1]: import requests

In [2]: s = requests.Session()

In [3]: r = s.post('http://127.0.0.1:8080')

In [4]: r.status_code
Out[4]: 200

In [5]: r = s.post('http://127.0.0.1:8080')

In [6]: r = s.post('http://127.0.0.1:8080')

In [7]: r.text
Out[7]: 'INSERTED A RANDOM STRING'

In [8]: r = s.get('http://127.0.0.1:8080')

In [9]: r.text
Out[9]: '3EF95BAf'
```

We update and delete every row just for a simplified learning sake.



```
In [16]: r = s.put('http://127.0.0.1:8080',{'another_string':"ARI"})
In [17]: r.text
Out[17]: 'UPDATED ALL ROWS'

In [18]: r = s.get('http://127.0.0.1:8080')
In [19]: r.text
Out[19]: 'ARI'
```

Dispatchers

A dispatcher is executed early during the request processing in order to determine which piece of code of your application will handle the incoming request.

Some Types : MethodDispatcher , RoutesDispatcher ,

Tools

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



Hosting An Application

We can host multiple applications using the `cherrypy.tree.mount()` method.

```
import cherryypy
class MyApp:
    @cherryypy.expose
    def index(self):
        return "MINE"

class YourApp:
    @cherryypy.expose
    def index(self):
        return "YOURS"

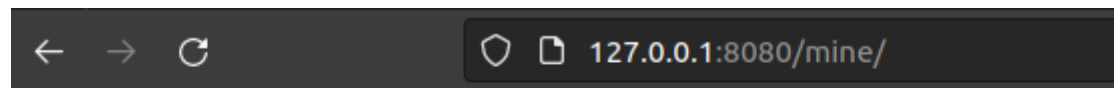
cherryypy.tree.mount(MyApp(), '/mine')
cherryypy.tree.mount(YourApp(), '/yours')

cherryypy.engine.start()
cherryypy.engine.block()
```



YOURS





MINE

Logging

There is a method `cherrypy.log()` for this logging purpose. CherryPy manages two loggers: (1) an access one that logs every incoming requests (2) an application/error log that traces errors or other application-level messages

```
import cherrypy
class MyApp:
    @cherrypy.expose
    def index(self):
        cherrypy.log("Hello There")

cherrypy.tree.mount(MyApp(), '/mine')
cherrypy.engine.start()
cherrypy.engine.block()
```

```
[09/Feb/2023:15:06:49] ENGINE Started monitor thread 'Autoreloader'.
[09/Feb/2023:15:06:49] ENGINE Serving on http://127.0.0.1:8080
[09/Feb/2023:15:06:49] ENGINE Bus STARTED
[09/Feb/2023:15:06:52] Hello There
127.0.0.1 - - [09/Feb/2023:15:06:52] "GET /mine/ HTTP/1.1" 200 - "" "Mozilla/5.0
20100101 Firefox/109.0"
```

It is able to log the errors also. `cherrypy.log(*Error*, traceback=True)` is used for that.



Disabling Logging

We hafta configure the application to be logging-disabled one.

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

Global Server Config

Configurations are generally settings added to the server. If we want to configure the server only to run on the port of 8080, we can go with,

```
cherrypy.config.update({'server.socket_port': 9090,  
                        'server.socket_host': '64.72.221.48'})
```

REMEMBER : It is only for configuring the server and the engine only! Not applications!



A Configuration may look like,

```
config = {'/':  
    {  
        'request.dispatch': cherrypy.dispatch.MethodDispatcher(),  
        'tools.json_out.on': True,  
    }  
}
```

Applications with same Config

```
cherrypy.tree.mount(Root1(), config="config.conf")  
cherrypy.tree.mount(Root2(), config="config.conf")
```

Applications with specific Config

```
cherrypy.tree.mount(Root1(), config="config1.conf")  
cherrypy.tree.mount(Root2(), config="config2.conf")
```



Multiple Parameters

```
import cherrypy

@cherrypy.expose
class NameWebService(object):
    string = ''

    def GET(self):
        return self.mystr

    @cherrypy.tools.accept(media='text/plain')
    def POST(self,*args):
        mystr = ''
        for i in args:
            mystr+=' '+str(i)
        self.mystr = mystr
        return mystr+" are given"

conf = {
    '/': {
        'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
        'tools.response_headers.on': True,
        'tools.response_headers.headers': [('Content-Type', 'text/plain')]
    }
}
cherrypy.quickstart(NameWebService(), '/', conf)
```

```
In [1]: import requests
```

```
In [2]: s = requests.Session()
```

```
In [3]: r = s.post('http://127.0.0.1:8080/ari/haran/sudhan')
```

```
In [4]: r.status_code
```

```
Out[4]: 200
```

```
In [5]: r = s.get('http://127.0.0.1:8080')
```

```
In [6]: r
```

```
Out[6]: <Response [200]>
```

```
In [7]: r.text
```

```
Out[7]: ' ari haran sudhan'
```



Sending and Receiving File

```
import cherrypy

class MyApp:
    @cherrypy.expose
    def index(self):
        return '''
            <form action=store method=GET>
                <input type='file' name='myFile'><br>
                <input type='submit'>
            </form>
        '''
    @cherrypy.expose
    def store(self,myFile):
        with open(myFile) as f:
            content = f.read()
        return content

cherrypy.quickstart(MyApp())
```

mine.txt

Hello GUYS! This is a txt file.

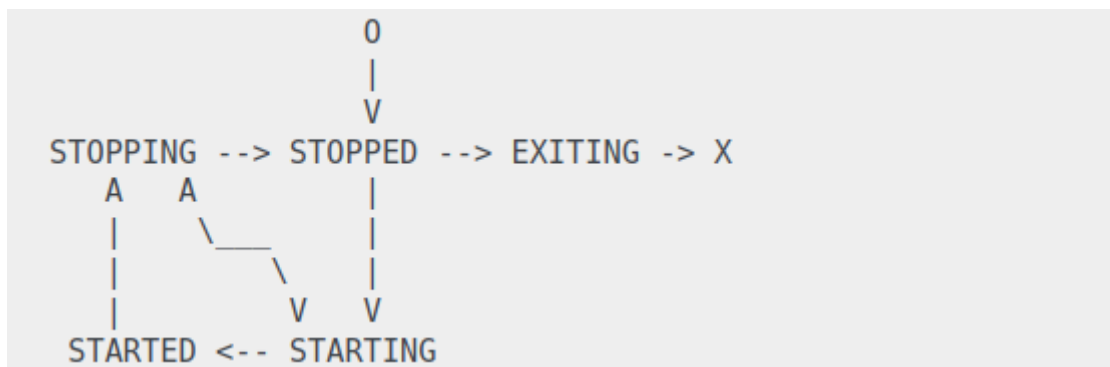


PubSub Pattern

CherryPy's backbone consists of a bus system implementing a simple publish/subscribe pattern. Simply put, in CherryPy everything is controlled via that bus. You can subscribe and publish to channels on a bus. A channel is bit like a unique identifier within the bus. When a message is published to a channel, the bus will dispatch the message to all subscribers for that channel. One interesting aspect of a pubsub pattern is that it promotes decoupling between a caller and the callee. A published message will eventually generate a response but the publisher does not know where that response came from. Thanks to that decoupling, a CherryPy application can easily access functionalities without having to hold a reference to the entity providing that functionality. Instead, the application simply publishes onto the bus and will receive the appropriate response, which is all that matter. As said earlier, CherryPy is



built around a pubsub bus. All entities that the framework manages at runtime are working on top of a single bus instance, which is named the engine. The bus implementation therefore provides a set of common channels which describe the application's lifecycle :



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

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

