PART 1 – SERVER

Some introductory RESTful[1] reading/watching:
```
http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
http://martinfowler.com/articles/richardsonMaturityModel.html
https://www.youtube.com/watch?v=pspy1H6A3FM
```

N.B. We are *not* implementing a full RESTful service in today's workshop (it's just RESTlike!)
Any service that **only** returns JSON is not RESTful... (see HATEOAS)
```
http://restcookbook.com/Mediatypes/json/
```
but RESTful type considerations will guide our interface design (e.g. the use of HTTP verbs)
(c.f. Richardson's level 2)
==========

We're using Tornado today because it's comparatively simple, but note, there are many other ways to create web services, e.g. Flask, node.js. Be sure to setup your virtual environment before you begin:
```
sudo apt-get update
sudo apt-get install python-pip python-virtualenv
virtualenv ENV
source ENV/bin/activate
pip install tornado
```

And, you will need to add the port numbers you use to the inbound security rules in Azure:
Azure: Resource groups:
  My-Resource-Group
    Icon that looks like a shield
      Inbound security rules
        + Add
          Name: MyTestServer
          Destination port range: 43210

1 A note on abbreviations: ReST = **Re**presentational **S**tate **T**ransfer, URI = **U**niform **R**esource **I**dentifier

STEP 1 - "HELLO WORLD" SERVER CODE
(create the following as "1.py", e.g. on the command line enter "nano 1.py")

```
$ nano 1.py

import tornado.httpserver
import tornado.ioloop
import tornado.web

class requestHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello Duncan")

application = tornado.web.Application([
    (r"/duncan.txt", requestHandler)
])

if __name__ == "__main__":
    http_server = tornado.httpserver.HTTPServer(application)
    http_server.listen(43210) #Use a sensible port number here*[2]
    tornado.ioloop.IOLoop.instance().start()
```

STEP 2 - RUN IT (be sure to set your python environment each time you login)

```
$ python 1.py
```

STEP 3 - TEST IT - Try your url in a browser,
`http://178.62.91.44:43210/duncan.txt`

Once this is working, copy this file and edit the new version as follow, i.e.

```
$ cp 1.py 2.py
$ nano 2.py
```

STEP 1 - Make the url match a regular expression
(For details on regex - see `http://en.wikipedia.org/wiki/Regular_expression`)

```
-     (r"/duncan.txt", requestHandler)
+     (r"/duncan/(.*)", requestHandler)
```

STEP 2 - Tell the callback to expect a second argument

```
-    def get(self):
+    def get(self,arg):
```

STEP 3 - Write out the arg to the web page

```
-        self.write("Hello Duncan")
+        self.write("Hello Duncan:"+arg)
```

STEP 4 - TEST IT, e.g.
`http://178.62.91.44:43210/duncan/testing123`

2 To choose a 'sensible' value for this see here:
`http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers`

OPTIONAL STEPS – THE 'SELF' OBJECT

Once all of the above is all working, copy this file and edit the new version, i.e.

```
$ cp 2.py 3.py
$ nano 3.py
```

Edit the get callback to dump the 'self' object to see what this requestHandler contains.

```
    def get(self,arg):
        for attr in dir(self):
            self.write("self.%s = %s<br>" % (attr, getattr(self, attr)))
```

...this is mostly ignorable, but there might be something useful in there that you could use later?

Encryption

As an aside: If you want to use https instead of http, you will need to generate a certificate/key.

```
$ mkdir keys
$ cd keys
$ openssl req -newkey rsa:2048 -new -nodes -x509 -days 3650 -keyout key.pem -out cert.pem
```

...and enter the required details following the prompts. This will create cert.pem and key.pem.
This is self certification (trusted certifiers charge money!), and so requires acceptance by user.

To get your server to use the new ssl credentials, make and edit 4.py

```
-    http_server = tornado.httpserver.HTTPServer(application)
+    http_server = tornado.httpserver.HTTPServer(application, ssl_options={
        "certfile": "keys/cert.pem",
        "keyfile": "keys/key.pem",
    })
```

TEST IT, (with https instead of http) e.g.

```
https://178.62.91.44:43210/duncan/testingssl.txt
```

...you will need to follow the prompts to accept your self-certified credentials.

Note, for the remaining examples I will not be using ssl, and you **must not** use it for your assessment (because the marking client will then not be able to access your server) but it's important you know how.

PART 2 - DATABASE

`sqlite implements a SQL database via local files and folders`
(since there is no 'server', this makes the database simple to setup and use, and is fine for small projects).
For info on SQL see `http://en.wikipedia.org/wiki/SQL`

STEP 1 - Start and interactive python shell
```
$ python
```

STEP 2 - Import the sqlite library
```
>>> import sqlite3
```

STEP 3 - Make (or open an existing database)
```
>>> db = sqlite3.connect('mytest.db')
```

STEP 4 - Create a cursor for this database
(see `http://en.wikipedia.org/wiki/Cursor_(databases)`)
```
>>> cursor = db.cursor()
```

STEP 5 - Create a table 'myData' containing integers 'myID'
```
>>> cursor.execute("CREATE TABLE myData (myID INT)")
```

STEP 6 - Add some data to this table, e.g.
```
>>> cursor.execute("INSERT INTO myData VALUES (56)")
>>> cursor.execute("INSERT INTO myData VALUES (1)")
>>> cursor.execute("INSERT INTO myData VALUES (99)")
>>> cursor.execute("INSERT INTO myData VALUES (20)")
```
(note, VALUES is plural, so a tuple, e.g. (a,b,c) hence the brackets even though our here data is singular)

STEP 7 - Write these changes to the db file (otherwise they are just in memory)
```
>>> db.commit()
```

STEP 8 - Read the data back...
```
>>> cursor.execute("SELECT * FROM myData")
>>> print cursor.fetchone()
>>> print cursor.fetchone()[0]
>>> for row in cursor:
...    print row
... (ctrl+D)
>>> print cursor.fetchone()
```

STEP 9 - Close the database and quit python
```
>>> db.close()
>>> quit()
```

STEP 10 - You should see the new database file in your directory
```
$ ls -l
```

STEP 11 - Access directly via sqlite3 (this is not via python but is good for debugging your database)
```
$ sudo apt-get install sqlite3
```
Load it into sqlite3 and take a look at the data
```
$ sqlite3 mytest.db
sqlite> .tables
...
sqlite> .schema
...
sqlite> select * from myData;
...etc...
sqlite> (ctrl+D)
```

PART 3 - SERVER AND DATABASE TOGETHER

5.py

Here is a (TERRIBLE!!) server that adds values to this table, then reads them back

```
import tornado.httpserver
import tornado.ioloop
import tornado.web
import sqlite3

_db   = sqlite3.connect('mytest.db')
_cursor = _db.cursor()

class getDataRequestHandler(tornado.web.RequestHandler):
    def get(self):
        _cursor.execute("SELECT * FROM myData")
        for row in _cursor:
            self.write(str(row[0])+"<br>")

class putDataRequestHandler(tornado.web.RequestHandler):
    def get(self):
        newValue = [int(self.get_argument("value"))]
        _cursor.execute("INSERT INTO myData VALUES (?)",newValue)
        _db.commit()
        self.write('OK')

application = tornado.web.Application([
    (r"/getdata", getDataRequestHandler),
    (r"/putdata", putDataRequestHandler),
])

if __name__ == "__main__":
    http_server = tornado.httpserver.HTTPServer(application)
    http_server.listen(43210)
    tornado.ioloop.IOLoop.instance().start()
```

TEST IT

```
http://178.62.91.44:43210/putdata?value=6
OK
http://178.62.91.44:43210/putdata?value=7
OK
http://178.62.91.44:43210/getdata
...
```

OK, this works! So why then is it such a TERRIBLE DESIGN!!?...