		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

Ejercicio 2

A continuación se muestran varios enlaces con información sobre la web y un tutorial para el desarrollo de una aplicación web genérica, con conexión a una base de datos y la utilización de plantillas para presentar la información.

- Python Web Framework `http://bottlepy.org/docs/dev/`
- World Wide Web consortium `http://www.w3.org`
- Web Design and Applications `http://www.w3.org/standards/webdesign/`

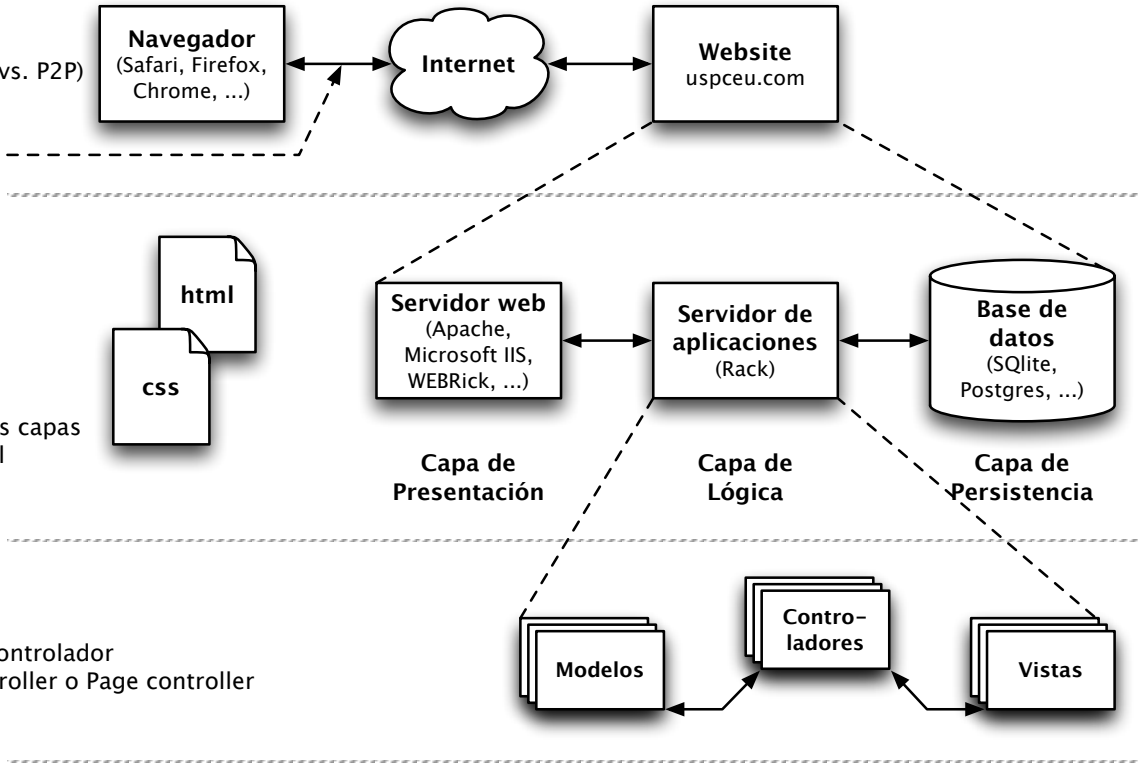
100.000 pies
- Cliente - servidor (vs. P2P)

50.000 pies
- HTTP y URIs

10.000 pies
- XHTML y CSS

5.000 pies
- Arquitectura de tres capas
- Escalado horizontal

1.000 pies
- Modelo - Vista - Controlador
- MVC vs. Front controller o Page controller



500 pies
- Modelos Active Record vs. Data Mapper
- Controladores RESTful (Representational State Transfer for self-contained actions)
- Template View vs. Transform View

- **Active Record**
- Data Mapper

- **REST**

- **Template View**
- Transform View

Bottle: Python Web Framework

Bottle is a fast, simple and lightweight `WSGI` micro web-framework for `Python`. It is distributed as a single file module and has no dependencies other than the `Python Standard Library`.

- **Routing**: Requests to function-call mapping with support for clean and dynamic URLs.
- **Templates**: Fast and pythonic built-in template engine and support for `mako`, `jinja2` and `cheetah` templates.
- **Utilities**: Convenient access to form data, file uploads, cookies, headers and other HTTP-related metadata.
- **Server**: Built-in HTTP development server and support for `paste`, `fapws3`, `bjoern`, `Google App Engine`, `cherrypy` or any other WSGI capable HTTP server.

		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

Example: “Hello World” in a bottle

```
from bottle import route, run, template

@route('/hello/<name>')
def index(name='World'):
    return template('<b>Hello {{name}}</b>!', name=name)

run(host='localhost', port=8080)
```

Run this script or paste it into a Python console, then point your browser to `http://localhost:8080/hello/world`. That’s it. Try different routes: `http://localhost:8080/hello/juan`

Tutorial: Todo-List Application

This tutorial should give a brief introduction to the Bottle WSGI Framework. The main goal is to be able, after reading through this tutorial, to create a project using Bottle. Within this document, not all abilities will be shown, but at least the main and important ones like routing, utilizing the Bottle template abilities to format output and handling GET / POST parameters.

To understand the content here, it is not necessary to have a basic knowledge of WSGI, as Bottle tries to keep WSGI away from the user anyway. You should have a fair understanding of the Python programming language. Furthermore, the example used in the tutorial retrieves and stores data in a SQL database, so a basic idea about SQL helps, but is not a must to understand the concepts of Bottle. Right here, SQLite is used. The output of Bottle sent to the browser is formatted in some examples by the help of HTML. Thus, a basic idea about the common HTML tags does help as well.

For the sake of introducing Bottle, the Python code “in between” is kept short, in order to keep the focus. Also all code within the tutorial is working fine, but you may not necessarily use it “in the wild”, e.g. on a public web server. In order to do so, you may add e.g. more error handling, protect the database with a password, test and escape the input etc.

1. Goals

At the end of this tutorial, we will have a simple, web-based ToDo list. The list contains a text (with max 100 characters) and a status (0 for closed, 1 for open) for each item. Through the web-based user interface, open items can be view and edited and new items can be added.

During development, all pages will be available on localhost only, but later on it will be shown how to adapt the application for a “real” server.

Bottle will do the routing and format the output, with the help of templates. The items of the list will be stored inside a SQLite database. Reading and writing the database will be done by Python code.

		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

We will end up with an application with the following pages and functionality:

- start page `http://localhost:8080/todo`
- adding new items to the list: `http://localhost:8080/new`
- page for editing items: `http://localhost:8080/edit/:no`
- catching errors

2. Before we start...

2.1. Install Bottle

Assuming that you have a fairly new installation of Python (version 2.5 or higher), you only need to install Bottle in addition to that. Bottle has no other dependencies than Python itself.

You can either manually install Bottle (download bottle.py file from http://bottlepy.org/bottle.py and save it into your project folder) or use Python's easy_install program: `easy_install bottle`

2.2. Further Software Necessities

As we use SQLite3 as a database, make sure it is installed. On Linux systems, most distributions have SQLite3 installed by default. SQLite is available for Windows and MacOS X as well and the sqlite3 module is part of the python standard library.

2.3. Create An SQL Database

First, we need to create the database we use later on. To do so, save the following script in your project directory as `todo-db.py` and run it with python, `python todo-db.py`. You can use the interactive interpreter too:

```
import sqlite3
con = sqlite3.connect('todo.db') # Warning: This file is created in the current directory
con.execute("CREATE TABLE todo (id INTEGER PRIMARY KEY, task char(100) NOT NULL, status bool NOT NULL)")
con.execute("INSERT INTO todo (task,status) VALUES ('Read A-byte-of-python to get a good introduction into Python',0)")
con.execute("INSERT INTO todo (task,status) VALUES ('Visit the Python website',1)")
con.execute("INSERT INTO todo (task,status) VALUES ('Test various editors for and check the syntax highlighting',1)")
con.execute("INSERT INTO todo (task,status) VALUES ('Choose your favorite WSGI-Framework',0)")
con.commit()
```

This generates a database-file `todo.db` with tables called todo and three columns id, task, and status. id is a unique id for each row, which is used later on to reference the rows. The column task holds the text which describes the task, it can be max 100 characters long. Finally, the column status is used to mark a task as open (value 1) or closed (value 0).

3. Using Bottle for a web-based ToDo list

Now it is time to introduce Bottle in order to create a web-based application. But first, we need to look into a basic concept of Bottle: routes.

3.1. Understanding routes

Basically, each page visible in the browser is dynamically generated when the page address is called. Thus, there is no static content. That is exactly what is called a "route" within Bottle: a certain address on the server. So, for example, when the page `http://localhost:8080/todo` is called from the browser, Bottle "grabs" the call and checks if there is any (Python) function defined for the route "todo". If so, Bottle will execute the corresponding Python code and return its result: `python todo.py`.

3.2. First Step - Showing All Open Items

So, after understanding the concept of routes, let's create the first one. The goal is to see all open items from the ToDo list:

		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

```
import sqlite3
from bottle import route, run

@route('/todo')
def todo_list():
    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT id, task FROM todo WHERE status LIKE '1'")
    result = c.fetchall()
    return str(result)

run()
```

Save the code a `todo.py`, preferably in the same directory as the file `todo.db`. Otherwise, you need to add the path to `todo.db` in the `sqlite3.connect()` statement.

Let's have a look what we just did: We imported the necessary module sqlite3 to access to SQLite database and from Bottle we imported `route` and `run`. The `run()` statement simply starts the web server included in Bottle. By default, the web server serves the pages on localhost and port 8080. Furthermore, we imported `route`, which is the function responsible for Bottle's routing. As you can see, we defined one function, `todo_list()`, with a few lines of code reading from the database. The important point is the decorator statement `@route('/todo')` right before the `def todo_list()` statement. By doing this, we bind this function to the route `/todo`, so every time the browsers calls `http://localhost:8080/todo`, Bottle returns the result of the function `todo_list()`. That is how routing within bottle works.

Actually you can bind more than one route to a function. So the following code:

```
@route('/todo')
@route('/my_todo_list')
def todo_list():
    ...
```

will work fine, too. What will not work is to bind one route to more than one function.

What you will see in the browser is what is returned, thus the value given by the `return` statement. In this example, we need to convert `result` in to a string by `str()`, as Bottle expects a string or a list of strings from the `return` statement. But here, the result of the database query is a list of tuples, which is the standard defined by the Python DB API.

Now, after understanding the little script above, it is time to execute it and watch the result yourself. Remember that on Linux- / Unix-based systems the file `todo.py` needs to be executable first. Then, just run `python todo.py` and call the page `http://localhost:8080/todo` in your browser. In case you made no mistake writing the script, the output should look like this:

```
[(2, u'Visit the Python website'), (3, u'Test various editors for and check the syntax highlighting')]
```

If so - congratulations! You are now a successful user of Bottle. In case it did not work and you need to make some changes to the script, remember to stop Bottle serving the page, otherwise the revised version will not be loaded.

Actually, the output is not really exciting nor nice to read. It is the raw result returned from the SQL query.

So, in the next step we format the output in a nicer way. But before we do that, we make our life easier.

3.3. Debugging and Auto-Reload

Maybe you already noticed that Bottle sends a short error message to the browser in case something within the script is wrong, e.g. the connection to the database is not working. For debugging purposes it is quite helpful to get more details. This can be easily achieved by adding the following statement to the script:

```
from bottle import run, route, debug
...
#add this at the very end:
debug(True)
run()
```

By enabling "debug", you will get a full stacktrace of the Python interpreter, which usually contains useful information for finding bugs. Furthermore, templates (see below) are not cached, thus changes to templates will take effect without stopping the server.

That debug(True) is supposed to be used for development only, it should not be used in production environments.

		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

Another quite nice feature is auto-reloading, which is enabled by modifying the run() statement to

```
run(reloader=True)
```

This will automatically detect changes to the script and reload the new version once it is called again, without the need to stop and start the server.

Again, the feature is mainly supposed to be used while developing, not on production systems.

3.4. Bottle Template To Format The Output

Now let's have a look at casting the output of the script into a proper format.

Actually Bottle expects to receive a string or a list of strings from a function and returns them by the help of the built-in server to the browser. Bottle does not bother about the content of the string itself, so it can be text formatted with HTML markup, too.

Bottle brings its own easy-to-use template engine with it. Templates are stored as separate files having a .tpl extension. The template can be called then from within a function. Templates can contain any type of text (which will be most likely HTML-markup mixed with Python statements). Furthermore, templates can take arguments, e.g. the result set of a database query, which will be then formatted nicely within the template.

Right here, we are going to cast the result of our query showing the open ToDo items into a simple table with two columns: the first column will contain the ID of the item, the second column the text. The result set is, as seen above, a list of tuples, each tuple contains one set of results.

To include the template in our example, just add the following lines:

```
from bottle import route, run, debug, template
...
result = c.fetchall()
c.close()
output = template('make_table', rows=result)
return output
...
```

So we do here two things: first, we import template from Bottle in order to be able to use templates. Second, we assign the output of the template make_table to the variable output, which is then returned. In addition to calling the template, we assign result, which we received from the database query, to the variable rows, which is later on used within the template. If necessary, you can assign more than one variable / value to a template.

Templates always return a list of strings, thus there is no need to convert anything. Of course, we can save one line of code by writing `return template('make_table', rows=result)`, which gives exactly the same result as above.

Now it is time to write the corresponding template, which looks like this:

```
%#template to generate a HTML table from a list of tuples (or list of lists, or tuple of tuples or ...)
<p>The open items are as follows:</p>
<table border="1">
%for row in rows:
  <tr>
  %for col in row:
    <td>{{col}}</td>
  %end
  </tr>
%end
</table>
```

Save the code as `make_table.tpl` in the same directory where `todo.py` is stored.

Let's have a look at the code: every line starting with % is interpreted as Python code. Please note that, of course, only valid Python statements are allowed, otherwise the template will raise an exception, just as any other Python code. The other lines are plain HTML markup.

As you can see, we use Python's `for` statement two times, in order to go through `rows`. As seen above, `rows` is a variable which holds the result of the database query, so it is a list of tuples. The first `for` statement accesses the tuples within the list, the second one the items within the tuple, which are put each into a cell of the table. It is important that you close all `for`, `if`, while etc. statements with `%end`, otherwise the output may not be what you expect.

		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

If you need to access a variable within a non-Python code line inside the template, you need to put it into double curly braces. This tells the template to insert the actual value of the variable right in place.

Run the script again and look at the output. Still not really nice, but at least more readable than the list of tuples. Of course, you can spice-up the very simple HTML markup above, e.g. by using in-line styles or `CSS` to get a better looking output.

3.5. Using GET and POST Values

As we can review all open items properly, we move to the next step, which is adding new items to the ToDo list. The new item should be received from a regular HTML-based form, which sends its data by the GET method.

To do so, we first add a new route to our script and tell the route that it should get GET data:

```
from bottle import route, run, debug, template, request
...
return template('make_table', rows=result)
...
@route('/new', method='GET')
def new_item():

    new = request.GET.get('task', '').strip()

    conn = sqlite3.connect('todo.db')
    c = conn.cursor()

    c.execute("INSERT INTO todo (task,status) VALUES (?,?)", (new,1))
    new_id = c.lastrowid

    conn.commit()
    c.close()

    return '<p>The new task was inserted into the database, the ID is %s</p>' % new_id
```

To access GET (or POST) data, we need to import `request` from Bottle. To assign the actual data to a variable, we use the statement `request.GET.get('task','').strip()` statement, where task is the name of the GET data we want to access. That's all. If your GET data has more than one variable, multiple `request.GET.get()` statements can be used and assigned to other variables.

The rest of this piece of code is just processing of the gained data: writing to the database, retrieve the corresponding id from the database and generate the output.

But where do we get the GET data from? Well, we can use a static HTML page holding the form. Or, what we do right now, is to use a template which is output when the route `/new` is called without GET data.

The code needs to be extended to:

```
...
@route('/new', method='GET')
def new_item():

    if request.GET.get('save','').strip():

        new = request.GET.get('task', '').strip()
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()

        c.execute("INSERT INTO todo (task,status) VALUES (?,?)", (new,1))
        new_id = c.lastrowid

        conn.commit()
        c.close()

        return '<p>The new task was inserted into the database, the ID is %s</p>' % new_id
    else:
        return template('new_task.tpl')
```

`new_task.tpl` looks like this:

```
<p>Add a new task to the ToDo list:</p>
<form action="/new" method="GET">
<input type="text" size="100" maxlength="100" name="task">
<input type="submit" name="save" value="save">
</form>
```

That's all. As you can see, the template is plain HTML this time.

		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

Now we are able to extend our to do list.

By the way, if you prefer to use POST data: this works exactly the same way, just use `request.POST.get()` instead.

3.6. Editing Existing Items

The last point to do is to enable editing of existing items.

By using only the routes we know so far it is possible, but may be quite tricky. But Bottle knows something called "dynamic routes", which makes this task quite easy.

The basic statement for a dynamic route looks like this:

```
@route('/myroute/:something')
```

The key point here is the colon. This tells Bottle to accept for `:something` any string up to the next slash. Furthermore, the value of `something` will be passed to the function assigned to that route, so the data can be processed within the function.

For our ToDo list, we will create a route `@route('/edit/:no)`, where no is the id of the item to edit.

The code looks like this:

```
@route('/edit/:no', method='GET')
def edit_item(no):

    if request.GET.get('save','').strip():
        edit = request.GET.get('task','').strip()
        status = request.GET.get('status','').strip()

        if status == 'open':
            status = 1
        else:
            status = 0

        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("UPDATE todo SET task = ?, status = ? WHERE id LIKE ?", (edit, status, no))
        conn.commit()

        return '<p>The item number %s was successfully updated</p>' % no
    else:
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("SELECT task FROM todo WHERE id LIKE ?", (str(no),))
        cur_data = c.fetchone()

        return template('edit_task', old=cur_data, no=no)
```

It is basically pretty much the same what we already did above when adding new items, like using GET data etc. The main addition here is using the dynamic route `:no`, which here passes the number to the corresponding function. As you can see, no is used within the function to access the right row of data within the database.

The template `edit_task.tpl` called within the function looks like this:

```
%#template for editing a task
%#the template expects to receive a value for "no" as well a "old", the text of the selected ToDo item
<p>Edit the task with ID = {{no}}</p>
<form action="/edit/{{no}}" method="get">
<input type="text" name="task" value="{{old[0]}}" size="100" maxlength="100">
<select name="status">
<option>open</option>
<option>closed</option>
</select>
<br/>
<input type="submit" name="save" value="save">
</form>
```

Again, this template is a mix of Python statements and HTML, as already explained above. A last word on dynamic routes: you can even use a regular expression for a dynamic route, as demonstrated later.

3.7. Dynamic Routes Using Regular Expressions

Bottle can also handle dynamic routes, where the "dynamic part" of the route can be a regular expression.

		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

So, just to demonstrate that, let's assume that all single items in our ToDo list should be accessible by their plain number, by a term like e.g. "item1". For obvious reasons, you do not want to create a route for every item. Furthermore, the simple dynamic routes do not work either, as part of the route, the term "item" is static. As said above, the solution is a regular expression:

```
@route('/item:item#[0-9]+#')
def show_item(item):
    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT task FROM todo WHERE id LIKE ?", (item))
    result = c.fetchall()
    c.close()
    if not result:
        return 'This item number does not exist!'
    else:
        return 'Task: %s' %result[0]
```

Of course, this example is somehow artificially constructed - it would be easier to use a plain dynamic route only combined with a validation. Nevertheless, we want to see how regular expression routes work: the line `@route(/item:item_#[0-9]+#)` starts like a normal route, but the part surrounded by # is interpreted as a regular expression, which is the dynamic part of the route. So in this case, we want to match any digit between 0 and 9. The following function "show_item" just checks whether the given item is present in the database or not. In case it is present, the corresponding text of the task is returned. As you can see, only the regular expression part of the route is passed forward. Furthermore, it is always forwarded as a string, even if it is a plain integer number, like in this case.

3.8. Returning Static Files

Sometimes it may become necessary to associate a route not to a Python function, but just return a static file. So if you have for example a help page for your application, you may want to return this page as plain HTML. This works as follows:

```
from bottle import route, run, debug, template, request, static_file

@route('/help')
def help():
    return static_file('help.html', root='/path/to/file')
```

At first, we need to import the static_file function from Bottle. As you can see, the return `static_file` statement replaces the `return` statement. It takes at least two arguments: the name of the file to be returned and the path to the file. Even if the file is in the same directory as your application, the path needs to be stated. But in this case, you can use `'.'` as a path, too. Bottle guesses the MIME-type of the file automatically, but in case you like to state it explicitly, add a third argument to `static_file`, which would be here `mimetype='text/html'`. `static_file` works with any type of route, including the dynamic ones.

3.9. Returning JSON Data

There may be cases where you do not want your application to generate the output directly, but return data to be processed further on, e.g. by JavaScript. For those cases, Bottle offers the possibility to return JSON objects, which is sort of standard for exchanging data between web applications. Furthermore, JSON can be processed by many programming languages, including Python

So, let's assume we want to return the data generated in the regular expression route example as a JSON object. The code looks like this:

```
@route('/json:json#[0-9]+#')
def show_json(json):
    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT task FROM todo WHERE id LIKE ?", (json))
    result = c.fetchall()
    c.close()
    if not result:
        return {'task':'This item number does not exist!'}
    else:
        return {'Task': result[0]}
```

As you can, that is fairly simple: just return a regular Python dictionary and Bottle will convert it automatically into a JSON object prior to sending. So if you e.g. call "`http://localhost/json1`" Bottle should in this case return the JSON object `{"Task": ["Read A-byte-of-python to get a good introduction into Python"]}`.

		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

3.10.Catching Errors

The next step may is to catch the error with Bottle itself, to keep away any type of error message from the user of your application. To do that, Bottle has an “error-route”, which can be a assigned to a HTML-error.

In our case, we want to catch a 403 error. The code is as follows:

```
from bottle import error

@error(403)
def mistake(code):
    return 'The parameter you passed has the wrong format!'
```

So, at first we need to import error from Bottle and define a route by error(403), which catches all “403 forbidden” errors. The function “mistake” is assigned to that. Please note that error() always passes the error-code to the function - even if you do not need it. Thus, the function always needs to accept one argument, otherwise it will not work.

Again, you can assign more than one error-route to a function, or catch various errors with one function each. So this code:

```
@error(404)
@error(403)
def mistake(code):
    return 'There is something wrong!'
```

works fine, the following one as well:

```
@error(403)
def mistake403(code):
    return 'The parameter you passed has the wrong format!'

@error(404)
def mistake404(code):
    return 'Sorry, this page does not exist!'
```

3.11.Summary

After going through all the sections above, you should have a brief understanding how the Bottle WSGI framework works. Furthermore you have all the knowledge necessary to use Bottle for your applications.

The following chapter give a short introduction how to adapt Bottle for larger projects. Furthermore, we will show how to operate Bottle with web servers which perform better on a higher load / more web traffic than the one we used so far.

4. Server setup

So far, we used the standard server used by Bottle, which is the WSGI reference Server shipped along with Python. Although this server is perfectly suitable for development purposes, it is not really suitable for larger applications. But before we have a look at the alternatives, let’s have a look how to tweak the settings of the standard server first.

4.1. Running Bottle on a different port and IP

As standard, Bottle serves the pages on the IP address `127.0.0.1`, also known as `localhost`, and on port `8080`. To modify the setting is pretty simple, as additional parameters can be passed to Bottle’s `run()` function to change the port and the address.

To change the port, just add `port=portnumber` to the run command. So, for example:

```
run(port=80)
```

would make Bottle listen to `port 80`.

To change the `IP address` where Bottle is listening:

```
run(host='123.45.67.89')
```

Of course, both parameters can be combined, like:

```
run(port=80, host='123.45.67.89')
```

The `port` and `host` parameter can also be applied when Bottle is running with a different server, as shown in the following section.

5. Final words

Now we are at the end of this introduction and tutorial to Bottle. We learned about the basic concepts of Bottle and wrote a first application using the Bottle framework.

As said in the introduction, this tutorial is not showing all shades and possibilities of Bottle. What we skipped here is e.g. receiving file objects and streams and how to handle authentication data. Furthermore, we did not show how templates can be called from within another template. For an introduction into those points, please refer to the full `Bottle documentation` .

6. Complete example listing

As the ToDo list example was developed piece by piece, here is the complete listing:

6.1. Listing for the HelloWebApp

Main code for the application file `hello.py`

```
from bottle import route, run, template

@route('/hello/<name>')
def index(name):
    return template('<b>Hello {{name}}</b>!', name=name)

@route('/bye')
def saybye():
    return '<b>Bye friend</b>!'

run(host='localhost', port=8080)
```

6.2. Listing for the DemoWebApp

Main code for the application file `todo.py`:

		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

```
import sqlite3
from bottle import route, run, debug, template, request, static_file, error

@route('/')
@route('/todo')
def todo_list():
    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT id, task FROM todo WHERE status LIKE '1';")
    result = c.fetchall()
    c.close()
    output = template('make_table', rows=result)
    return output

@route('/new', method='GET')
def new_item():
    if request.GET.get('save','').strip():
        new = request.GET.get('task', '').strip()
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("INSERT INTO todo (task,status) VALUES (?,?)", (new,1))
        new_id = c.lastrowid
        conn.commit()
        c.close()
        return '<p>The new task was inserted into the database, the ID is %s</p><p><a href="/">Show all tasks...</a></p>'
% new_id
    else:
        return template('new_task.tpl')

@route('/edit/:no', method='GET')
def edit_item(no):
    if request.GET.get('save','').strip():
        edit = request.GET.get('task','').strip()
        status = request.GET.get('status','').strip()
        if status == 'open':
            status = 1
        else:
            status = 0
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("UPDATE todo SET task = ?, status = ? WHERE id LIKE ?", (edit,status,no))
        conn.commit()
        return '<p>The item number %s was successfully updated</p><p><a href="/">Show all tasks...</a></p>' %no
    else:
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("SELECT task FROM todo WHERE id LIKE ?", (str(no),))
        cur_data = c.fetchone()
        return template('edit_task', old = cur_data, no = no)

@route('/item:item#[0-9]+#')
def show_item(item):
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("SELECT task FROM todo WHERE id LIKE ?", (item,))
        result = c.fetchone()
        c.close()
        if not result:
            return '<p>This item number does not exist!</p><p><a href="/">Show all tasks...</a></p>'
        else:
            return 'Task: %s <p><a href="/">Show all tasks...</a></p>' %result[0]

@route('/help')
def help():
    return static_file('help.html', root='.')

@error(403)
def mistake403(code):
    return '<p>There is a mistake in your url!</p>'

@error(404)
def mistake404(code):
    return '<p>Sorry, this page does not exist!</p>'

debug(True)
#remember to remove reloader=True option and debug(True) when you move your application from development to a productive
environment
run(host='localhost', port = 8080)
```

		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

Template `make_table.tpl`:

```
%#template to generate a HTML table from a list of tuples (or list of lists, or tuple of tuples or ...)
<p>The open items are as follows:</p>
<table border="1">
%for row in rows:
  <tr>
  %for col in row:
    <td>{{col}}</td>
  %end
  </tr>
%end
</table>
```

Template `edit_task.tpl`:

```
%#template for editing a task
%#the template expects to receive a value for "no" as well a "old", the text of the selected ToDo item
<p>Edit the task with ID = {{no}}</p>
<form action="/edit/{{no}}" method="get">
<input type="text" name="task" value="{{old[0]}}" size="100" maxlength="100">
<select name="status">
<option>open</option>
<option>closed</option>
</select>
<br/>
<input type="submit" name="save" value="save">
</form>
```

Template `new_task.tpl`:

```
%#template for the form for a new task
<p>Add a new task to the ToDo list:</p>
<form action="/new" method="GET">
<input type="text" size="100" maxlength="100" name="task">
<input type="submit" name="save" value="save">
</form>
```

Help file `help.html`

```
<html>
<head>
  <title>This is the help</title>
</head>
<body>
  <h1>Help</h1>
  <p>TBD.</p>
</body>
</html>
```

		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

Entregable y tareas opcionales

Después de seguir el tutorial, los alumnos deberán enviar sus trabajos usando el campus online como archivo adjunto en un fichero ZIP (archivo comprimido) antes de la fecha de entrega a la siguiente dirección: `jsanz.eps@ceu.es`. El fichero de base de datos así como el script de creación también deberá ser incluido. La forma más sencilla consiste en comprimir la carpeta que contiene todo el trabajo realizado incluido el informe con las respuestas.

Preguntas de repaso sobre protocolos de red y desarrollo básico de aplicaciones web

1. Al acceder desde el navegador a la aplicación “Hello World” utilizando la url `http://localhost:8080/hello/world` sin arrancar el servidor, ¿qué error se muestra en el navegador y por qué?
2. Si con el servidor arrancado, accedemos desde el navegador a la url `http://localhost:8080/hello/world`, ¿qué aparece en pantalla y por qué? ¿qué información se muestra en la consola del servidor?
3. Una vez arrancado el servidor de la aplicación web “Hello world” en la forma descrita en este documento, si accedemos desde el navegador a la url `http://localhost:8008/hello/world` ¿qué error se muestra en el navegador y por qué?
4. Si con el servidor arrancado, accedemos desde el navegador a la url `http://localhost:8080/bye`, ¿qué error aparece en pantalla y por qué? ¿qué información se muestra en la consola del servidor?
5. Parar el servidor para crear una nueva ruta/recurso y configurar la respuesta del servidor. Abrimos el fichero de la aplicación y justo debajo de la ruta existente copiar el código que se muestra a continuación, guardar el fichero y construir de nuevo la aplicación para arrancar el servidor y escuchar de nuevo las peticiones.

```
@route('/bye')
def saybye():
    return '<b>Bye friend</b>!'
```

6. Comprobar desde el navegador que esposible acceder ahora a la ruta especificada sin errores y que se muestra un resultado en pantalla. ¿Qué diferencias puedes comentar respecto a la definición de la ruta y configuración de la acción entre ambas rutas del fichero?
7. Se pide asimismo escribir un pequeño informe en el que se permita comprobar cómo se realiza la conversación entre el cliente y el servidor (ciclo petición-respuesta) en una página web de su elección (online edition newspaper, ...) utilizando el inspector web. Para ver el menú de desarrollo en Safari puede ser necesario activarlo desde la pestaña de configuración avanzada en el menú preferencias de la aplicación:
    1. Los tiempos de carga y tamaño de la respuesta enviada por el servidor
    2. Los encabezados HTTP de la petición
    3. Los encabezados HTTP de la respuesta
    4. Información de los recursos adicionales transferidos para completar la petición.
8. Comparar dicho informe con los resultados ofrecidos por el framework Bottle y el inspector web del cliente en la aplicación `hellowebapp`.
9. Antes de continuar comprobar que la aplicación `hellowebapp` no está funcionando para evitar problemas con el servidor y el puerto de escucha. Arrancar el servidor como se ha indicado anteriormente para la otra aplicación entregada `demowebapp` y acceder desde el navegador a la url `http://localhost:8080`. ¿qué aparece en pantalla y por qué? ¿qué información se muestra en la consola del servidor?
10. Parar el servidor y crear la estructura de base de datos necesaria. Para ello se va a ejecutar el fichero `todo-db.py`. Una vez creada la estructura de base de datos es posible arrancar de nuevo el servidor y comprobar que es posible acceder a la aplicación con normalidad. ¿Podrías intentar describir el contenido de dicho fichero?

		ASIGNATURA tecnologías para la programación y el diseño web i	CURSO 2	GRUPO 01
CALIFICACION	EVALUACION	APELLIDOS	NOMBRE	DNI
		OBSERVACIONES Tecnologías para el desarrollo web	FECHA 24/02/2016	FECHA ENTREGA 18/03/2016

11. Si accedemos a la dirección `http://localhost:8080/todo` y analizamos con el inspector web el resultado que se muestra en pantalla. ¿Qué diferencias existen entre la respuesta enviada por el servidor y la respuesta construida por el navegador (árbol DOM)?

12. Algunas tareas que deberán ser desarrolladas por los alumnos en la aplicación web de tareas y que se podrán incluir en la entrega para su valoración en la entrea final se muestran a continuación como fichero comprimido en formato ZIP. Para realizar estas tareas puede ser necesario parar el servidor y volverlo a arrancar para que la configuración del servidor sea efectiva. Leer la documentación proporcionada por Bottle en el sitio oficial:

    1. Definición de varias rutas para la misma página y comprobar que funcionan
    2. Definición de enlaces para mejorar la navegación entre páginas/acciones: por ejemplo, trate de pensar qué ocurre si al editar una tarea, quiero cancelar la acción.
    3. Definición de nuevas rutas y configurar las acciones para aumentar la funcionalidad de la aplicación: ver tareas completadas en lugar de tareas pendientes.