Temasek Junior College

2024 JC2 H2 Computing

Web Applications 7 – HTTP Responses, Templates and Rendering

Section	4	Computer Networks
Unit	4.2	Web Applications
Objectives	4.2.3	Use HTML, CSS (for clients) and Python (for the server) to create a web application that is able to: - accept user input (text and image file uploads) - process the input on the local server - store and retrieve data using an SQL database - display the output (as formatted text/ images/table)
	4.2.4	Test a web application on a local server.

1 HTTP Responses and Status Codes

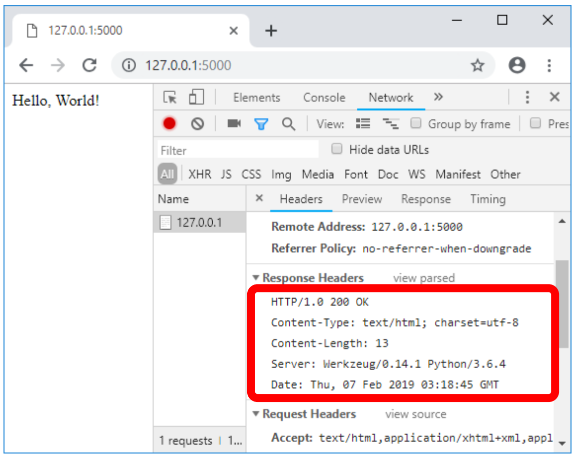
Thus far, we have being writing functions that return short strings that appear to display without problems in the web browser.

However, Flask has actually been prepending various headers behind the scenes to produce valid HTTP responses.

Exercise 1
Try running the following code on both IDLE (as **helloworld.py**) and Jupyter Notebook.

```python
import flask

app = flask.Flask(__name__)

@app.route('/')
def index():
  return 'Hello, World!'

if __name__ == '__main__':
  app.run()
```

With Google Chrome's Developer Tools opened, visit **http://127.0.0.1:5000/** and examine the response headers. You should see that Flask has actually added various headers to form a complete HTTP response.



```
Remote Address: 127.0.0.1:5000
Referrer Policy: no-referrer-when-downgrade
▾ Response Headers    view parsed
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 13
Server: Werkzeug/0.14.1 Python/3.6.4
Date: Thu, 07 Feb 2019 03:18:45 GMT
▾ Request Headers    view source
Accept: text/html,application/xhtml+xml,appl
```

Together with the content, the complete HTTP response sent to your browser is similar to the following:

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 13
Server: Werkzeug/0.14.1 Python/3.7.0
Date: Tue, 22 Jan 2019 08:01:52 GMT

Hello, World!
```

Notice that Flask assumes that our output has a Content-Type of **`"text/html"`**.

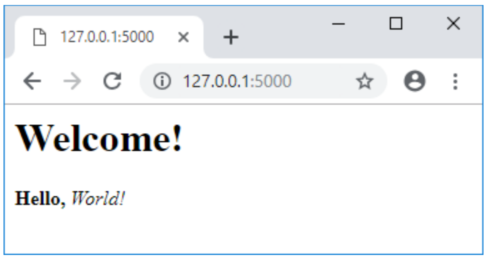
This means that Flask actually expects our function to return a full HTML document and not just a plain string.

However, most browsers are very forgiving and will treat our string as a snippet of HTML intended for the document's **`<body>`**.

Exercise 2
Try running the following code on both IDLE (as **`return_html.py`**) and Jupyter Notebook.

```python
import flask

app = flask.Flask(__name__)

@app.route('/')
def index():
  return '<h1>Welcome!</h1> <b>Hello,</b> <i>World!</i>'

if __name__ == '__main__':
  app.run()
```

Exercise 2 shows that HTML tags can be used in our return value. Notice that the tags have been appropriately interpreted has HTML.


127.0.0.1:5000

127.0.0.1:5000

Welcome!

Hello, *World!*

However, be aware that this is not correct practice. In fact, the functions mapped to the routes should really return full HTML documents.

1.1 Changing the Status Code

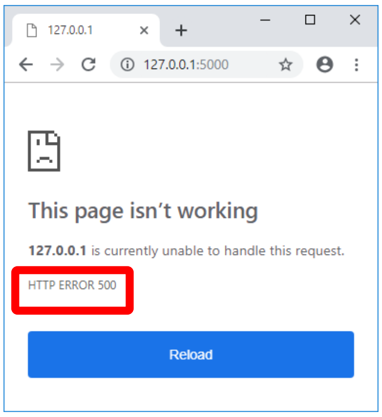
By default, Flask also assumes that our responses have a HTTP status code of **`200 (OK)`**. This is usually what we want, but if needed we can override this by returning a tuple instead of just a string. The replacement HTTP status code should be provided as the second item of the tuple.

Exercise 3
Try running the following code on both IDLE (as **`return_status.py`**) and Jupyter Notebook.

```
1   import flask
2   
3   app = flask.Flask(__name__)
4   
5   @app.route('/')
6   def index():
7     return ('', 500)
8   
9   if __name__ == '__main__':
10    app.run()
```

Line **`7`** in **Exercise 3** demonstrates the use of a tuple to return the status code **`500`**, which represents an Internal Server Error.

If we run this program and try to visit **`http://127.0.0.1:5000/`**, we should be greeted with the status code **`HTTP 500 (Internal Server Error)`**:



This page isn't working

127.0.0.1 is currently unable to handle this request.

HTTP ERROR 500

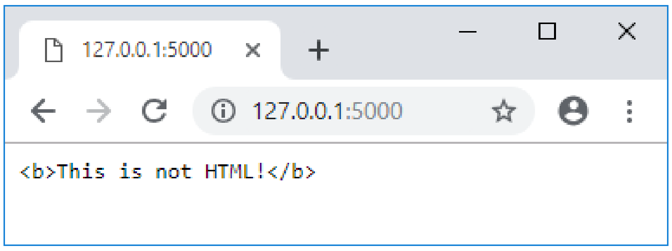
1.2 Changing the Response Headers

We can also add additional response headers by putting them into a Python dictionary and returning this dictionary as the third item of our return tuple. For instance, if we want the web browser to treat our response as plain text instead of HTML, we can replace the Content-Type header value with **`"text/plain"`** instead.

Exercise 14
Try running the following code on both IDLE (as **`plain_text.py`**) and Jupyter Notebook.

```
import flask

app = flask.Flask(__name__)

@app.route('/')
def index():
  headers = {'Content-Type': 'text/plain'}
  return ('<b>This is not HTML!</b>', 200, headers)

if __name__ == '__main__':
  app.run()
```

In **Exercise 4**, visiting **`http://127.0.0.1:5000/`** demonstrates that the string we return is no longer treated as HTML by the browser:



```
<b>This is not HTML!</b>
```

1.3 Redirecting to Another URL

Besides overriding the HTTP status code and response headers, Flask also lets us generate a response that tells the web browser to load a different URL instead. This is called a **redirect** and is useful when the location of a document has moved or when we want to let another Flask route take over the handling of a request.

To perform a redirect, import the **`redirect()`** function from the **`flask`** module and call it with the destination URL or path as the first argument. Then, use the response generated by **`redirect()`** as the return value of the function.

Exercise 5
Try running the following code on both IDLE (as **`redirect_ext.py`**) and Jupyter Notebook.

```
import flask
from flask import redirect

app = flask.Flask(__name__)

@app.route('/')
def index():
  return redirect('http://example.com')

if __name__ == '__main__':
  app.run()
```

Instead of redirecting to an external site, we usually want to redirect the user to another of our routes instead. In such cases, we should use **`url_for()`** to look up the correct path based on the function that we want to reach:

Exercise 6
Try running the following code on both IDLE (as **`redirect_int.py`**) and Jupyter Notebook.

```
1   import flask
2   from flask import redirect, url_for
3
4   app = flask.Flask(__name__)
5
6   @app.route('/new_url/')
7   def moved_index():
8     return 'You have reached the new URL!'
9
10  @app.route('/')
11  def index():
12    return redirect(url_for('moved_index'))
13
14  if __name__ == '__main__':
15    app.run()
```

Notice how **`url_for()`** is used to look up the path for **`redirect()`** on line **12** instead of hardcoding the redirected path as a string.

2 Templates and Rendering

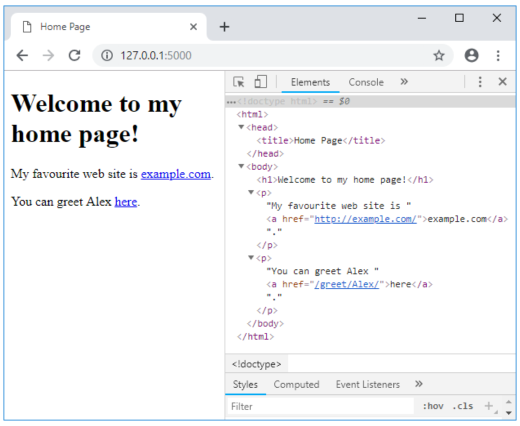
Instead of returning short HTML snippets or redirecting users, we can return full HTML documents complete with headings and hyperlinks in our Flask application.

Exercise 7
Try running the following code on both IDLE (as **`response_without_templates.py`**) and Jupyter Notebook.

```
import flask

app = flask.Flask(__name__)

@app.route('/')
def home():
  html = '<!DOCTYPE html>\n<html>'
  html += '<head><title>Home Page</title></head>'
  html += '<body><h1>Welcome to my home page!</h1>'
  html += '<p>My favourite web site is '
  html += '<a href="http://example.com/">'
  html += 'example.com</a>.</p>'
  html += '<p>You can greet Alex '
  html += '<a href="/greet/Alex/">here</a>.</p>'
  html += '</body></html>'
  return html

@app.route('/greet/<name>/')
def greet(name):
  html = '<!DOCTYPE html>\n<html>'
  html += '<head><title>Greetings!</title></head>'
  html += '<body><h1>Hello, {}!</h1>'.format(name)
  html += '</body></html>'
  return html

if __name__ == '__main__':
  app.run()
```

Upon visiting **`http://127.0.0.1:5000/`**, a full HTML page appears, complete with page title, headings and hyperlinks.

The View Source feature **`Ctrl`** + **`U`** can be used to verify that the HTML for the page comes directly from the **`home()`** function of the above Python program.



Welcome to my home page!

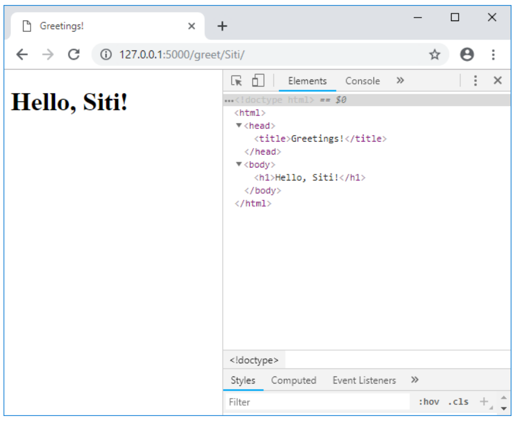
My favourite web site is example.com.

You can greet Alex here.

```
...<!doctype html> == $0
<html>
 ▼<head>
    <title>Home Page</title>
  </head>
 ▼<body>
    <h1>Welcome to my home page!</h1>
   ▼<p>
      "My favourite web site is "
      <a href="http://example.com/">example.com</a>
      "."
    </p>
   ▼<p>
      "You can greet Alex "
      <a href="/greet/Alex/">here</a>
      "."
    </p>
  </body>
</html>
```

<!doctype>

Visit **`http://127.0.0.1:5000/greet/Siti/`**. The result has a page title and the greeting for Siti is formatted to look like a heading using the **`<h1>`** tag.



```
...<!doctype html> == $0
  <html>
  ▼<head>
      <title>Greetings!</title>
    </head>
  ▼<body>
      <h1>Hello, Siti!</h1>
    </body>
  </html>
```

<!doctype>

4.1 The Need for Templates

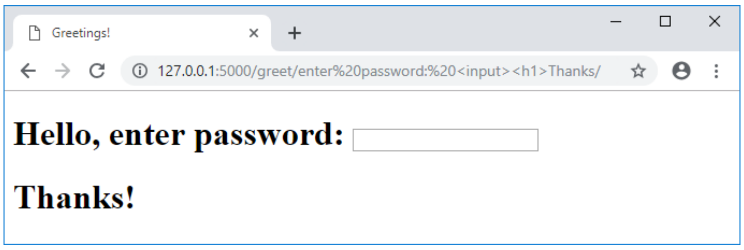
Generating dynamic HTML content the way shown in **Exercise 7** is powerful but dangerous.

Notice that in the `greet_name()` function we inserted the `name` variable into our output without checking its contents. This lets a malicious user inject his or her own code into the output and potentially cause all kinds of mischief.

For instance, the following link we bring users to what appears to be a legitimate form asking for the user's password.

```
127.0.0.1:5000/greet/enter%20password:%20<input><h1>Thanks/
```

However, in reality, the form does not come from our application at all and was injected into the page from HTML code in the URL's path.



Hello, enter password:

Thanks!

Besides possible security issues, constructing HTML documents by joining Python strings can also become quite messy. For instance, it is easy to be confused between the use of HTML and Python in the `home()` and `greet_name()` functions.

This is why, in practice, we usually do not generate HTML responses by manually manipulating strings in Python code. Instead, we put the HTML content in a separate file called a **template** with placeholders for where the dynamic content should be inserted.

When we need to output HTML, we use a **template engine** to load the template and fill in the placeholders. The template engine also helps to escape special characters such as `<` and `>` when filling in the placeholders so that HTML injection such as the case above is avoided. This process of filling in the placeholders to produce the final HTML that is used for the response is called **rendering**.

4.2 The Jinja2 Template Engine

Flask provides a built-in template engine named **Jinja2**.

By default, Flask expects all templates to be located in a subfolder named **`templates`**.

To start using Jinja2, create a subfolder named **`templates`** in the folder where your Flask programs are stored, then save the following file in this folder as **`home.html`**:

Exercise 8
Create the following template using Notepad++ and save it as **`templates/home.html`**

```
<!DOCTYPE html>
<html>
  <head>
    <title>Home Page</title>
  </head>
  <body>
    <h1>Welcome to my home page!</h1>
    <p>My favourite web site is
      <a href="http://example.com/">example.com</a>.
    </p>
    <p>You can greet Alex
      <a href="{{ url_for('greet', name='Alex') }}">here</a>.
    </p>
  </body>
</html>
```

Exercise 9
Create the following template using Notepad++ and save it as **`templates/greet.html`**

```
<!DOCTYPE html>
<html>
  <head>
     <title>Greetings!</title>
  </head>
  <body>
    <h1>Hello, {{ visitor }}!</h1>
  </body>
</html>
```

In a Jinja2 template, placeholders are surrounded by double braces and have the form **`{{ expression }}`** where **`expression`** is a **Jinja2 expression**.

When using templates, be careful not to confuse Jinja2 expressions with Python expressions. Although they are very similar, Jinja2 expressions and Python expressions have different syntaxes and operate in different environments.

For instance, **`len()`** and many other standard Python functions are not available in Jinja2 expressions. You will learn more about the differences between the two as you progress.

Recall that the process of filling in the placeholders of a template is called **rendering**. The task of rendering a template is typically performed by a function named **`render_template()`** that can be imported from the **`flask`** module.

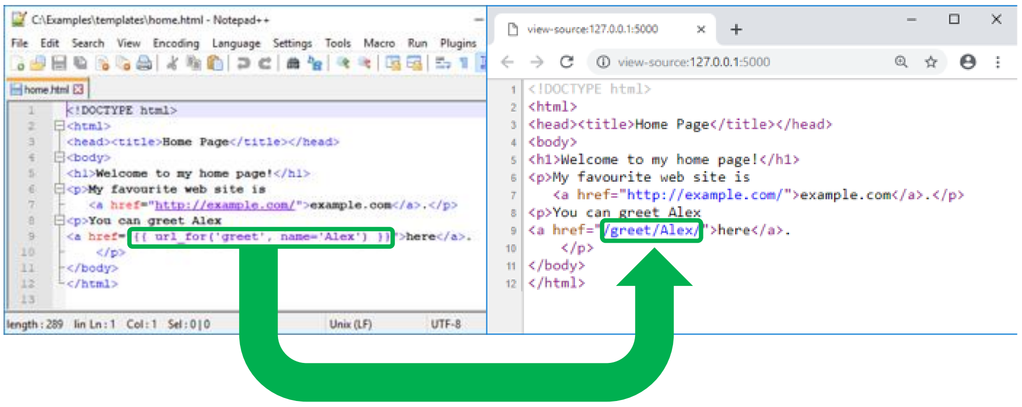
Exercise 10
Try running the following code on both IDLE (as **`response_with_templates.py`**) and Jupyter Notebook.

```
import flask
from flask import render_template

app = flask.Flask(__name__)

@app.route('/')
def home():
  return render_template('home.html')

@app.route('/greet/<name>/')
def greet(name):
  return render_template('greet.html', visitor=name)

if __name__ == '__main__':
  app.run()
```

Upon visiting **`http://127.0.0.1:5000/`**, view the source by pressing `Ctrl` **+** `U` to verify that the template is indeed rendered and no placeholders are present.

Click the link to visit **`http://127.0.0.1:5000/greet/Alex/`** and verify that there are no placeholders in its HTML source as well.

Template `home.html` Source of `http://127.0.0.1:5000/`


C:\Examples\templates\home.html - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins
home.html
length : 289 lin Ln : 1 Col : 1 Sel : 0 | 0 Unix (LF) UTF-8
view-source:127.0.0.1:5000

```
<!DOCTYPE html>
<html>
<head><title>Home Page</title></head>
<body>
<h1>Welcome to my home page!</h1>
<p>My favourite web site is
   <a href="http://example.com/">example.com</a>.</p>
<p>You can greet Alex
<a href="{{ url_for('greet', name='Alex') }}">here</a>.
    </p>
</body>
</html>
```

```
<!DOCTYPE html>
<html>
<head><title>Home Page</title></head>
<body>
<h1>Welcome to my home page!</h1>
<p>My favourite web site is
   <a href="http://example.com/">example.com</a>.</p>
<p>You can greet Alex
<a href="/greet/Alex/">here</a>.
    </p>
</body>
</html>
```

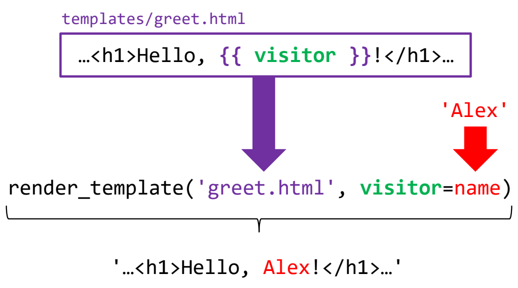
Rendered HTML is same as template except placeholders are replaced with the result of Jinja2 expressions

4.3 Passing Values to Templates

The **`render_template()`** function accepts the name of a template in the **`templates`** subfolder as its first argument, followed by 0 or more keyword arguments assigning values to Jinja2 variables that may be used by the template.

For instance, in **Exercise 10**, line **`12`** of the Python code renders a template named **`greet.html`** and specifies that within the template, the value of **`name`** should be assigned to a Jinja2 variable named **`visitor`**.

This corresponds to line **`5`** of the HTML template **`greet.html`** where the Jinja2 placeholder **`{{ visitor }}`** is replaced by this value.

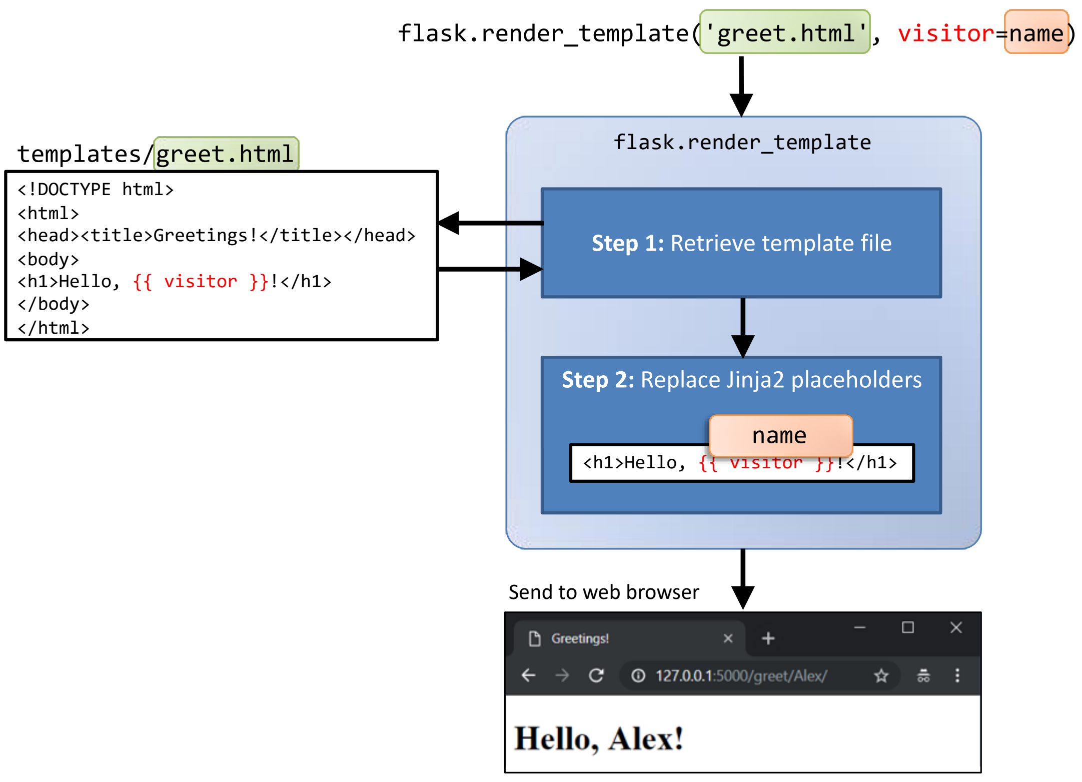


```
templates/greet.html
…<h1>Hello, {{ visitor }}!</h1>…

'Alex'

render_template('greet.html', visitor=name)

'…<h1>Hello, Alex!</h1>…'
```

Do not confuse Jinja2 variables with Python variables.

If **`render_template()`** is called without any keyword arguments, values from the Python environment are NOT passed to the template. To use Python values in the generated HTML, we must pass them over as keyword arguments when **`render_template()`** is called.

The following diagram illustrates how **`render_template()`** retrieves a template file and replaces its placeholders to produce the final HTML that is sent to the browser:



```
flask.render_template('greet.html', visitor=name)
```

```
templates/greet.html
<!DOCTYPE html>
<html>
<head><title>Greetings!</title></head>
<body>
<h1>Hello, {{ visitor }}!</h1>
</body>
</html>
```

flask.render_template
Step 1: Retrieve template file
Step 2: Replace Jinja2 placeholders
name
<h1>Hello, {{ visitor }}!</h1>
Send to web browser
Greetings!
127.0.0.1:5000/greet/Alex/

Hello, Alex!

Exercise 11A
Create the following four templates and save them in a **`templates/`** subfolder.

`templates/A.html`

```
<!doctype html>
<html>
  <head>
    <title>A</title>
  </head>
  <body>
    A
  </body>
</html>
```

`templates/B.html`

```
<!doctype html>
<html>
  <head>
    <title>B</title>
  </head>
  <body>
    B
  </body>
</html>
```

`templates/C.html`

```
<!doctype html>
<html>
  <head>
    <title>C</title>
  </head>
  <body>
    C
  </body>
</html>
```

`templates/D.html`

```
<!doctype html>
<html>
  <head>
    <title>D</title>
  </head>
  <body>
    D
  </body>
</html>
```

Each template in Exercise 21A is meant to be shown for a different URL:

URL	Template to use
`http://127.0.0.1:5000/A`	`A.html`
`http://127.0.0.1:5000/B`	`B.html`
`http://127.0.0.1:5000/C`	`C.html`
`http://127.0.0.1:5000/D`	`D.html`

Exercise 11B
Complete the code for **app.py** shown below. Lines **4**, **8**, **12**, and **16** are missing

```
1   import flask
2   app = flask.Flask(__name__)
3
4
5   def view_A():
6     return flask.render_template('A.html')
7
8
9   def view_B():
10    return flask.render_template('B.html')
11
12
13  def view_C():
14    return flask.render_template('C.html')
15
16
17  def view_D():
18    return flask.render_template('D.html')
19
20  if __name__ == '__main__':
21    app.run()
```

Exercise 11C
Generalise Python program code and HTML templates so that only one template is needed.

template.html (Complete lines **5** and **9**)

```
1   <!doctype html>
2   <html>
3     <head>
4       <title>
5
6       </title>
7     </head>
8     <body>
9
10    </body>
11  </html>
```

app.py (Complete lines **4** to **6**)

```
1   import flask
2   app = flask.Flask(__name__)
3
4
5   def view(      ):
6
7
8   if __name__ == '__main__':
9       app.run()
```

Notice that this application allows other combinations of URL, for example:
http://127.0.0.1:5000/BBB and **http://127.0.0.1:5000/q**.

[Challenge] Edit the program so only the four cases given are allowed and error code **500** is returned for other cases.

4.4 Avoiding Hardcoded Links

Although most Python functions are not available from Jinja2, **`render_template()`** automatically includes Flask's **`url_for()`** function so paths can be generated based on the Python function that needs to be called instead being hardcoded. This is most useful for creating HTML links, as demonstrated by the following lines in **Exercise 8**:

```
<p>You can greet Alex
    <a href="{{ url_for('greet', name='Alex') }}">here</a>.
```

Using **`url_for()`** is more future-proof than hardcoding the path like this:

```
<p>You can greet Alex <a href="/greet/Alex/">here</a>.
```

While hardcoded links are shorter, they need to be updated every time we change the path that is routed with a function.

Using **`url_for()`** lets us update our paths without changing every template that links to it.

Exercise 22
Create the following program **`app.py.`** Thereafter, use the **`url_for()`** function to complete the **`links.html`** template so that each link leads the respective message.

`app.py`

```
import flask
app = flask.Flask(__name__)

@app.route('/')
def home():
  return flask.render_template('links.html')

@app.route('/greeting')
def hello():
  return 'Hello, World!'

@app.route('/<int:year>')
def report(year):
  return 'year is ' + str(year)

if __name__ == '__main__':
  app.run()
```

`templates/links.html`

```
<!doctype html>
<html>
  <head>
    <title>Links</title>
  </head>
  <body>
    <p><a href="{{ url_for('hello') }}">Hello, World!</a></p>
    <p><a href="{{ url_for('report', year=2023) }}">2023</a></p>
  </body>
</html>
```