



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: <ul style="list-style-type: none"> - 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 been writing functions which return short strings that appear to display without problems in the web browser.

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

Exercise 1

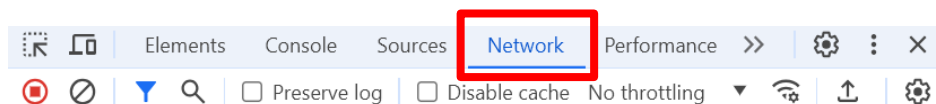
Try running the following code on both IDLE (as `helloworld.py`) and Jupyter Notebook.

```

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

1. On Google Chrome, access the Developer Tools using **Ctrl** + **Shift** + **I**.

2. Within the Develop Tools menu, select Network.

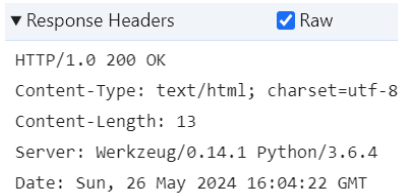


3. Visit `http://127.0.0.1:5000/`.

4. A document with name `127.0.0.1` will be generated in the Developer Tools panel. Click on the document.

Name	Status	Type	Initiator	Size	Time
127.0.0.1	200	docume...	Other	167 B	28 ms

5. Under the Response Headers section, select Raw.



```

▼ Response Headers ☒ Raw
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: Sun, 26 May 2024 16:04:22 GMT

```

6. Examine the script generated. You should see that Flask has actually added various headers to form a complete HTTP response.
7. 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/3.6.4
Date: Sun, 26 May 2024 16:04:22 GMT

Hello, World!

```

Notice that Flask assumes that the output has a Content-Type of `text/html`. This means that Flask actually expects the `index()` function to return a full HTML document and not just a plain string. Nevertheless, most browsers are very “forgiving”, and will treat the return string as a snippet of HTML intended for the document's `<body>`.

Exercise 2

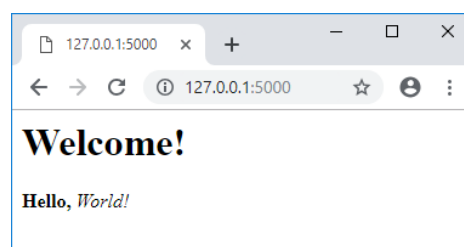
Try running the following code on both IDLE (as `return_html.py`) and Jupyter Notebook.

```

1 import flask
2
3 app = flask.Flask(__name__)
4
5 @app.route('/')
6 def index():
7     return '<h1>Welcome!</h1> <b>Hello,</b> <i>World!</i>'
8
9 if __name__ == '__main__':
10     app.run()

```

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



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

1.1 Changing the Status Code

Flask assumes that responses have a HTTP status code of **200 (OK)**. Nevertheless, the status code can be overridden with the use of a tuple containing an empty string as the first item. 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

Reload

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 the return tuple. Recall that the second item should be the HTTP status code.

For instance, if we want the web browser to treat the response as plain text instead of HTML, we can replace the **Content-Type** header value with **"text/plain"** instead of **"text/html"**.

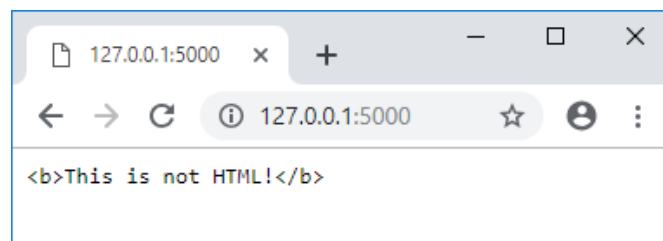
Exercise 4

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

```
1 import flask
2
3 app = flask.Flask(__name__)
4
5 @app.route('/')
6 def index():
7     headers = {'Content-Type': 'text/plain'}
8     return ('<b>This is not HTML!</b>', 200, headers)
9
10 if __name__ == '__main__':
11     app.run()
```

Line 7 in **Exercise 4** demonstrates the use of a dictionary to specify the **Content-Type** as **"text/plain"** and Line 8 shows the use of this dictionary as the third item of the return tuple to modify the response headers.

Visiting `http://127.0.0.1:5000/` demonstrates that the string returned is no longer treated as HTML by the browser:



Notice that **** and **** are no longer processed as HTML tags.

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.

```
1 import flask
2 from flask import redirect
3
4 app = flask.Flask(__name__)
5
6 @app.route('/')
7 def index():
8     return redirect('http://example.com')
9
10 if __name__ == '__main__':
11     app.run()
```

Instead of redirecting to an external site, we usually want to redirect the user to another path internally. 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('/')
7 def index():
8     return redirect(url_for('moved_index'))
9
10 @app.route('/new_url/')
11 def moved_index():
12     return 'You have reached the new URL!'
13
14 if __name__ == '__main__':
15     app.run()
```

Notice how `url_for()` is used to look up the path for `redirect()` on line 8 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.

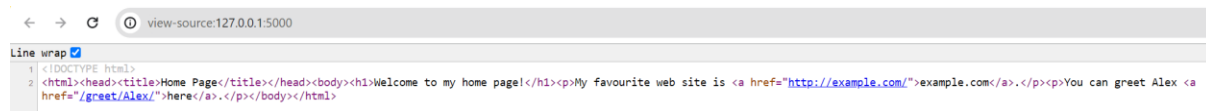
Exercise 7

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

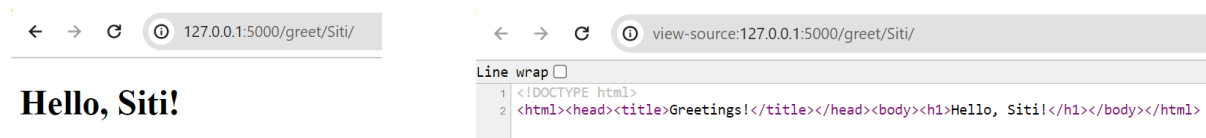
```
1 import flask
2
3 app = flask.Flask(__name__)
4
5 @app.route('/')
6 def home():
7     html = '<!DOCTYPE html>\n<html>'
8     html += '<head><title>Home Page</title></head>'
9     html += '<body><h1>Welcome to my home page!</h1>'
10    html += '<p>My favourite web site is '
11    html += '<a href="http://example.com/">'
12    html += 'example.com</a>.</p>'
13    html += '<p>You can greet Alex '
14    html += '<a href="/greet/Alex/">here</a>.</p>'
15    html += '</body></html>'
16    return html
17
18 @app.route('/greet/<name>/')
19 def greet(<name>):
20     html = '<!DOCTYPE html>\n<html>'
21     html += '<head><title>Greetings!</title></head>'
22     html += f'<body><h1>Hello, {<name>}!</h1>'
23     html += '</body></html>'
24     return html
25
26 if __name__ == '__main__':
27     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.



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.



2.1 The Need for Templates

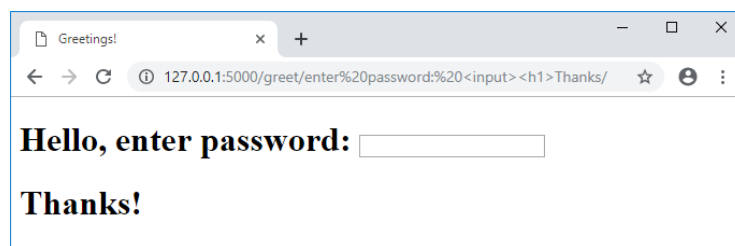
Generating dynamic HTML content the way shown in **Exercise 7** is powerful but dangerous.

Notice that in the `greet_name()` function, the `name` variable was inserted into the output without checking its contents. This gives rise to the risk of HTML injection. Malicious users can inject their own code into the output and potentially cause all kinds of mischief.

For instance, the following link will 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<h1>Thanks/

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



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 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**.

2.2 The Jinja2 Template Engine

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

By default, Flask expects all HTML 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 required HTML templates in the **templates** folder.

2.2.1 Jinja2 Expressions

Exercise 8

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

```

1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Home Page</title>
5    </head>
6    <body>
7      <h1>Welcome to my home page!</h1>
8      <p>My favourite web site is
9        <a href="http://example.com/">example.com</a>.
10     </p>
11     <p>You can greet Alex
12       <a href="{{ url_for('greet', name='Alex') }}">here</a>.
13     </p>
14   </body>
15 </html>

```

Exercise 9

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

```

1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Greetings!</title>
5    </head>
6    <body>
7      <h1>Hello, {{ visitor }}!</h1>
8    </body>
9  </html>

```

As mentioned in **Section 2.1**, HTML content should be stored in a template file, separate from the Python Flask application. The template file can contain placeholders where dynamic content can be inserted. The files created in **Exercises 8** and **9** are two such template files.

In Line 12 of **Exercise 8** and Line 7 of **Exercise 9**, expressions enclosed in double braces have been written. They are `{{ url_for('greet', name='Alex') }}` and `{{ visitor }}`. These are **Jinja2 expressions**, which are of the form `{{ expression }}`. Jinja2 expressions are used as placeholders in HTML templates where dynamic content is to be inserted.

When using HTML 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 as Jinja2 expressions.

Despite so, Jinja expressions offer a multitude of functionalities which include but are not limited to the following:

- access to variables passed from the Flask application to the HTML template.
- use of Jinja2 or Flask built-in functions such as `url_for()` to generate URLs or perform basic logic.
- Implementation of control flow structures such as the `if` conditional and `for` loop within the HTML template.

We shall now attempt to create a Flask application that makes use of the templates created in **Exercises 8** and **9**.

2.2.2 Rendering with render_template()

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.

```
1 import flask
2 from flask import render_template
3
4 app = flask.Flask(__name__)
5
6 @app.route('/')
7 def home():
8     return render_template('home.html')
9
10 @app.route('/greet/<name>/')
11 def greet(name):
12     return render_template('greet.html', visitor=name)
13
14 if __name__ == '__main__':
15     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.



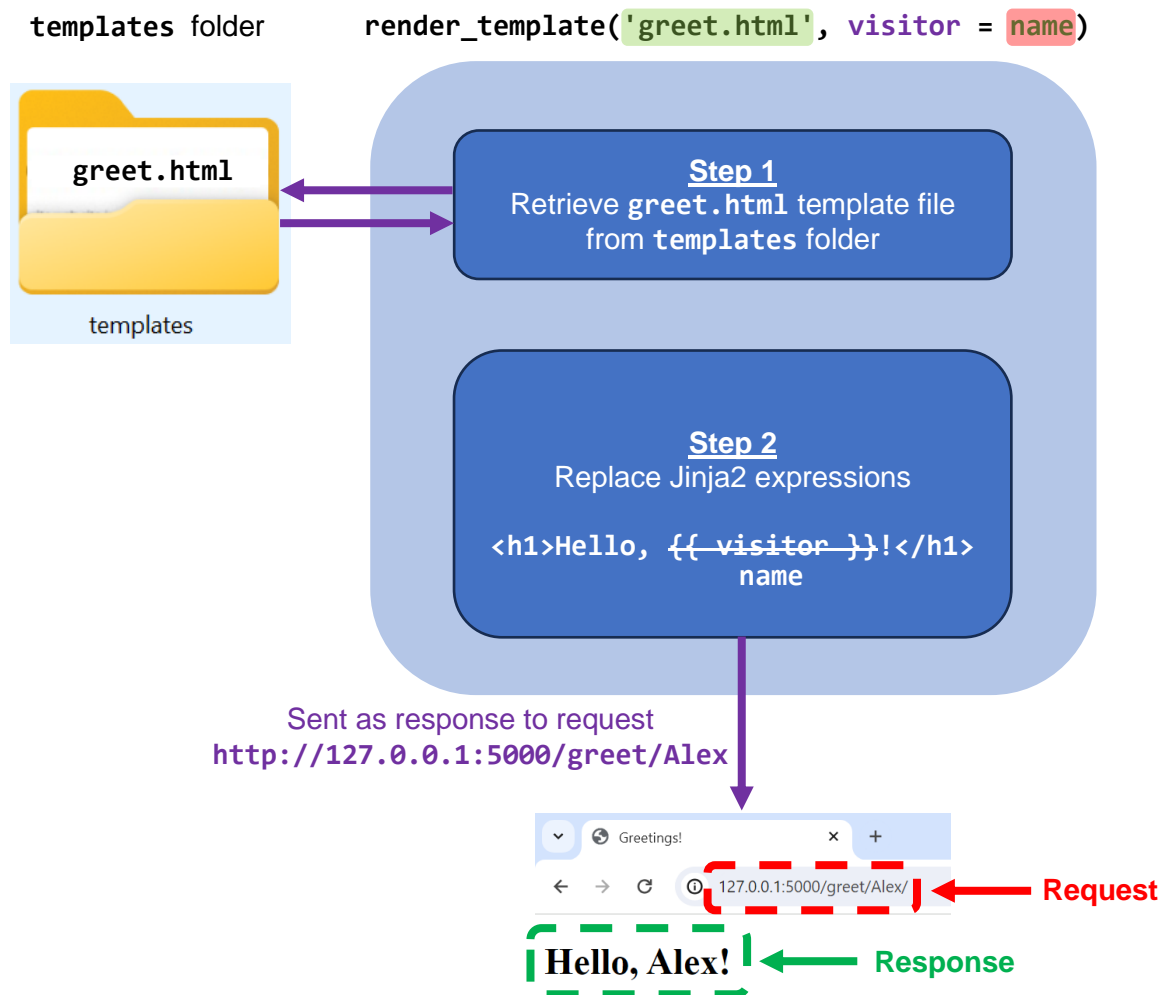
The HTML script of `http://127.0.0.1:5000/` no longer contains the Jinja2 expression `{{ url_for('greet', name='Alex') }}`. The expression has been replaced by the actual value `/greet/Alex/`. This is possible as the Jinja2 template engine interprets `url_for()` in the Jinja2 expression as the Flask built-in utility `url_for()`, allowing the Jinja2 expression to be evaluated with the path of the `greet()` function in the Flask application.

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.

Essentially, when a page is requested from the Python Flask application, the corresponding HTML template will be loaded. Flask then uses the Jinja2 template engine to process the HTML template. This involves evaluating all the placeholder expressions within the template.

Once the template engine finishes processing all the placeholder expressions and replacing them with their evaluated values, the final HTML script which contains both the static content that is already written in the template and the dynamically generated content from the placeholder expressions, will be sent by the Python Flask application as the response to the request.

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:

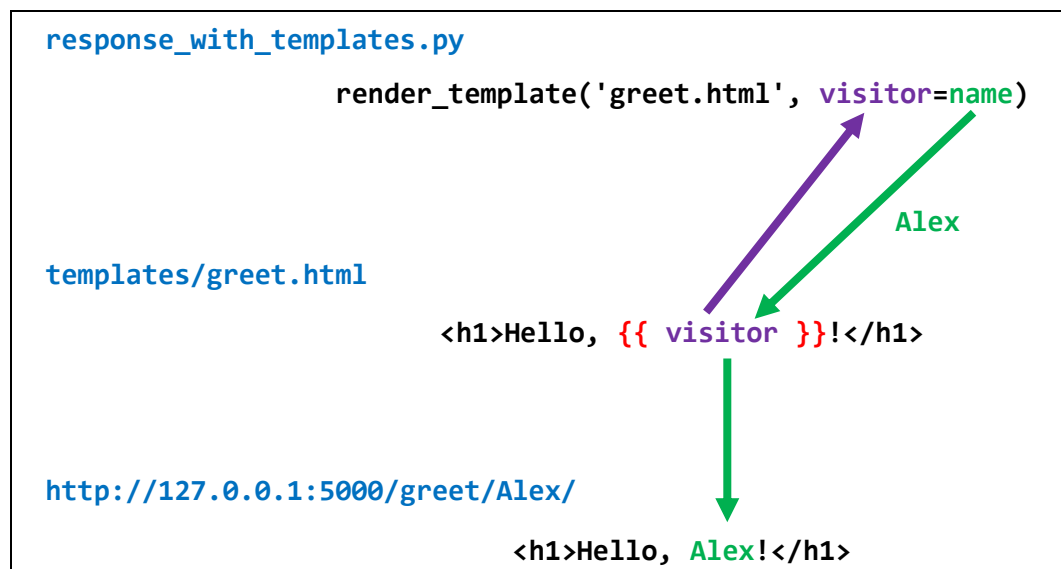


2.2.3 Passing Values to Templates

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 7 of the HTML template `greet.html` created in **Exercise 9**, where the Jinja2 expression `{{ visitor }}` is replaced by the value of `name` in the Python code.



If `render_template()` is called without any keyword arguments i.e. the only argument provided is the file name of the HTML template, values from the Python environment WILL NOT be passed to the template.

Therefore, to use Python values in the generated HTML, it is a must to pass them over as keyword arguments when `render_template()` is called. Herein, it is important to note that while doing so, we should not confuse Jinja2 variables with Python variables. The keyword argument is always specified in the format `Jinja2_variable = Python variable`.

Exercise 11A

Create the following four templates and save them in a **templates/** subfolder.

templates/A.html

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

templates/B.html

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

templates/C.html

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

templates/D.html

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

Each template in **Exercise 11A** 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

The code for Flask application to be used with the HTML templates create in **Exercise 11A** is provided below.

Exercise 11B

Complete the code for `app.py` shown below. Lines 4, 8, 12, and 16 are missing

```

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

Notice that writing the code in **Exercise 11B** is cumbersome as it involves defining four almost identical functions (with the exception of the letter and template used). The same goes for the creation of the four templates in **Exercise 11A**. The HTML script is almost identical (with the exception of the letter).

Imagine developing a webpage that needs to display content in a particular format, but yet the content changes dynamically based on the user request. It might be impossible to predict all possible requests and consequently impossible to create all necessary HTML templates and write all required functions in the Flask application.

This problem can be circumvented altogether by adopting the method of passing values from the Flask application to the HTML templates via Jinja2 expressions.

Exercise 11C presents a re-design of the HTML scripts in **Exercise 11A** and Flask Application in **Exercise 11B** to demonstrate how this can be done.

Exercise 11C

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

templates/letter.html (Complete lines 5 and 9)

```

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

```

alphabet.py (Complete lines 4 to 6)

```

1 import flask
2 from flask import render_template
3
4 app = flask.Flask(__name__)
5
6 @app.route('/<alphabet>')
7 def view(alphabet):
8     return render_template('letter.html', letter=alphabet)
9
10 if __name__ == '__main__':
11     app.run()

```

The four HTML templates in **Exercise 11A** has now been reduced to a single HTML template that can display content dynamically depending on the value of **letter**. This value will be that of **alphabet**, that is passed over from the Flask application (see Line 6 of **alphabet.py**)

Notice that this application now allows other combinations of URL i.e. beyond the four letters A, B, C and D, 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.

[Solution]

alphabet_500.py

```

1 import flask
2 from flask import render_template
3
4 app = flask.Flask(__name__)
5
6 @app.route('/<alphabet>')
7 def view(alphabet):
8     if alphabet in ['A', 'B', 'C', 'D']:
9         return render_template('letter.html', letter=alphabet)
10    else:
11        return ('', 500)
12
13 if __name__ == '__main__':
14     app.run()

```

4.4 Avoiding Hardcoded Links

Although most Python functions are not available from Jinja2, `render_template()` automatically includes Flask's `url_for()` utility, allowing paths to be generated based on the Python function that needs to be called. This eliminates the need to hardcode a path. 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>.
</p>
```

Using `url_for()` is more future-proof than hardcoding the path as follows:

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

While hardcoded links are shorter, they need to be updated every time we change the path that is mapped to a function. Using `url_for()` lets us update our paths without changing every template that links to it.

Exercise 12

Create the following program `app_12.py`. Thereafter, use the `url_for()` function to complete the `links.html` template so that each link leads the respective message.

app_12.py

```
1 import flask
2 from flask import render_template
3
4 app = flask.Flask(__name__)
5
6 @app.route('/')
7 def home():
8     return render_template('links.html')
9
10 @app.route('/greeting')
11 def hello():
12     return 'Hello, World!'
13
14 @app.route('/<int:year>')
15 def report(year):
16     return 'year is ' + str(year)
17
18 if __name__ == '__main__':
19     app.run()
```


templates/links.html	
1	<!doctype html>
2	<html>
3	<head>
4	<title>Links</title>
5	</head>
6	<body>
7	<p>Hello, World!</p>
8	<p>2023</p>
9	</body>
10	</html>