Chapter 8: Web applications (Jinja) Part 2

Contents

Syllabus Learning Outcomes

4.2 Web Applications
Understand the concepts and techniques for developing web applications.

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 What is jinja?

Jinja is a popular and powerful template engine for Python web development. It is designed to be easy to use and flexible, allowing developers to build complex and dynamic web applications with ease. Jinja provides a simple syntax for creating templates, which can be used to generate HTML, XML, and other markup languages. With Jinja, developers can separate the presentation layer from the business logic, making it easier to maintain and update their web applications. Jinja is also extensible, allowing developers to create their own custom filters and extensions to further enhance their templates.

2 Jinja Syntax

Jinja has been introduced in Web Applications (Flask) part 1. Now, let's delve deeper into the various elements that make up the Jinja syntax, which are utilized for defining and manipulating templates.

Type	Syntax	Description
Variables	`{{ }}`	Variables in Jinja are defined using the `{{ }}` tags. They are used to output dynamic content in the template. Example: `{{ name }} in greet.html`

Control Structures	`{% if condition %}`
	`.....`
	`{% else %}`
	`.....`
	`{% endif %}`
	
	`{% for item in items %}`
	`...`
	`{% endfor %}`
		In each of these examples, the control structure is defined using opening and closing tags, such as `{% if %}` and `{% endif %}` for if/else statements, and `{% for %}` and `{% endfor %}` for loops. The content to be executed within the control structure is then placed between these tags.

Comments	`{# #}`	Comments in Jinja are define using the `{# #}` tags. They are used to add comments to the template that are ignored during rendering.
Inheritance	`{% extends %}`	Inheritance in Jinja allows templates to inherit content from a base template. This is done using the `{% extends %}` tag. Example: `{% extends "layout.html" %}`
Filters	|	Filters in Jinja are used to modify the output of a variable. They are defined using the | character.

Exercise 1

The exercise1.py application is designed to receive a name as a parameter from the URL and display a greeting in the browser with the format "Hello, [name]!".

Complete the code in hello.html file and run the application to view the output in the browser.

exercise1.py

```
01 from flask import Flask, render_template
02
03 app = Flask(__name__)
04
05 @app.route('/<my_name>')
06 def index(my_name):
07     return render_template('hello.html', name=my_name)
08
09 if __name__ == '__main__':
10     app.run()
```

hello.html

```
01 <!DOCTYPE HTML>
02 <html>
03   <head>
04     <title> {{ _________ }} </title>
05   </head>
06   <body>
07     <h1>Hello, {{ _________ }}!</h1>
08   </body>
09 </html>
10
```

3 Using url_for

In Flask, url_for() is a built-in function that is used to generate URLs based on the Python function name and its arguments. It takes the name of a function as its first argument, followed by any arguments required by that function. The url_for() function then returns a URL that corresponds to the specified function and arguments.

Using url_for() in your Flask application is recommended over hardcoding URLs in your templates, because it allows you to change the URL routes in your application without having to update every template that references them. Additionally, url_for() can automatically handle URL encoding of arguments, making it easier to pass special characters or non-ASCII text in URLs.

Let's take a look at how we can implement url_for in index.html from the previous notes.

app.py

```
from flask import Flask, render_template, request

app = Flask(__name__)

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

@app.route('/greet', methods=['POST'])
def greet():
    name = request.form["name"]
    return render_template("greet.html", name=name)

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

index.html referencing using decorator

```
<!DOCTYPE html>

<html>
  <head>
    <title>hello</title>
  </head>
  <body>
    <form action="/greet" method="post">
      <input name="name" placeholder="Name" type="text">
      <button type="submit">Greet</button>
    </form>
  </body>
</html>
```

When using Flask, a function decorated with the @app.route() decorator maps to a URL route and returns a response that is sent to the client. Flask uses the URL path of an incoming request to determine which function should handle the request, and that function then generates a response which is returned to the client.

In the above index.html file, the URL route /greet is mapped to a response, in app.py, that retrieves input from the user, renders the greet.html template, and passes the input name as an argument to be displayed in the browser.

index.html using url_for()

```
<!DOCTYPE html>

<html>
  <head>
    <title>hello</title>
  </head>
  <body>
    <form action="{{ url_for("greet") }}" method="post">
      <input name="name" placeholder="Name" type="text">
      <button type="submit">Greet</button>
    </form>
  </body>
</html>
```

However, to avoid hardcoding, we can use the url_for() function to generate the appropriate URL. In order to use url_for() in our HTML form, we need to set the action attribute to `action="{{ url_for("function_name") }}"`, where function_name is the name of the Flask function that we want to submit the form data to.

For example, if we have a function named `greet()` that we want to submit the form data to, we will set the form's action attribute to `action="{{ url_for("greet") }}"`. This will generate the appropriate URL for the greet function, which will be used when the form is submitted.

Exercise 2

Complete the code for links.html using url_for().

links.py

```
01 from flask import Flask, render_template
02
03 app = Flask(__name__)
04
05 @app.route('/')
06 def home():
07     return render_template('links.html')
08
09 @app.route('/greeting')
10 def hello():
11     return '<b>Hello World!</b>'
12
13 @app.route('/<year>')
14 def which_year(year):
15     return 'The year is {}.'.format(year)
16
17 if __name__ == '__main__':
18     app.run()
```

links.html

```
01 <!DOCTYPE html>
02 <html>
03   <head>
04     <title>Links</title>
05   </head>
06   <body>
07     <p><a href= "{{ _____ }}">Hello World!</a></p>
08
09     <p><a href= "{{ _____ }}">Year</a></p>
10   </body>
11 </html>
12
```

Note that `url_for` is a Python function in Flask that is used to generate URLs dynamically. It is not a HTML statement, so in order for HTML to recognize it, we use Jinja syntax in our templates. Jinja is a templating language that is integrated into Flask, and it allows us to embed dynamic content, including `url_for` function calls, into our HTML templates.

4 Filters

In Jinja, filters are functions that can be applied to variables in templates to modify their values or formatting. Filters are applied to variables using the | symbol followed by the name of the filter and any arguments it may require.

There are many built-in filters available in Jinja, including lower, title, trim, striptags, urlencode, default, date, length, safe and many others. In addition, you can define your own custom filters in your Flask application using the app.template_filter() decorator.

For example, the upper filter can be used to convert a string to uppercase:

```
{{ "hello" | upper }}
```

This will output hello in uppercase: HELLO.

The safe filter is used in Jinja to indicate that a string should be output as-is, without being escaped. This is useful if you want to include HTML or other code in your templates that you want to be rendered as HTML or code, rather than being treated as plain text. Below is an example of a template that uses the safe filter:

Let's say we have a Flask application that has a route that renders a template with a string variable `message`. Without the safe filter, any HTML tags in the `message` variable would be escaped and displayed as plain text. To display the HTML tags as actual tags, we can use the safe filter.

Safe filter example

```
01 <!DOCTYPE html>
02 <html>
03   <head>
04     <title>Safe Filter Example</title>
05   </head>
06   <body>
07     <h1>{{ message | safe }}</h1>
08   </body>
09 </html>
```

In this example, the `message` variable contains HTML tags. By using the safe filter, we tell Jinja to render the `message` variable as HTML rather than plain text. The resulting HTML will render the `message` variable as expected, with the HTML tags included and interpreted by the browser.

It is important to note that using the safe filter can be a security risk if the content you are outputting contains user-supplied data, since it can allow malicious code to be injected into your page. Therefore, you should always use the safe filter with caution, and make sure that you are sanitizing any user input before outputting it as HTML.

Exercise 3

length.py

```
01  from flask import Flask, render_template
02
03  app = Flask(__name__)
04
05  @app.route('/<name>')
06  def length(name):
07      return render_template('length.html', name=name)
08
09  if __name__ == '__main__':
10      app.run()
11
```

length.html

```
01  <!DOCTYPE HTML>
02
03  <html>
04    <head>
05      <title>Length of Name</title>
06    </head>
07
08    <body>
09      <h1>Length of Name</h1>
10      <!-- insert your code below -->
11    </body>
    </html>
```

Complete length.html template which takes in a name using app.route and output “Hello <name>, your name is <length of name> characters long.”

5 Conditional statements

5.1 If statements

The Jinja if statement is a conditional statement used to control which parts of a template should be included or excluded during rendering. When a part of the template is excluded, it will not be rendered. However, unlike Python, Jinja does not group code blocks by indentation. Therefore, it is necessary to include an endif statement to indicate where the if statement ends. Moreover, because the if, elif, and else clauses are marked by separate {% and %} blocks, colons are not needed in Jinja.

length_if.py

```
01 from flask import Flask, render_template, request
02
03 app = Flask(__name__)
04
05 @app.route('/', methods=['GET','POST'])
06 def length():
07     if request.method == "GET":
08         return render_template("length_if.html")
09     else:
10         name = request.form["name"]
11         return render_template("length_if.html", name=name)
12
13 if __name__ == '__main__':
14     app.run()
```

length.html

```
01 <!DOCTYPE HTML>
02
03 <html>
04   <head>
05     <title>Length of Name</title>
06   </head>
07
08   <body>
09     <h1>Length of Name</h1>
10
11     <form method='post' action="{{ url_for('length') }}">
12       <label for='name'>Enter your name: </label>
13       <input type='text' id='name' name='name'>
14       <br>
15       <input type='submit' class='button' value='Submit'>
16     </form>
17     <br><br>
18
19     {% if name %}
20       Hello {{ name }}, your name is {{ name | length }} characters long.
21     {% endif %}
22   </body>
23 </html>
24
```

The length.html template includes a conditional statement that checks if the variable 'name' has a value, and if so, it will display a message to the user using the entered name and the length of the name. If not, the browser will only display the text box for user to type in a name.

Exercise 4

results.py

```
01  from flask import Flask, render_template, request
02
03  app = Flask(__name__)
04
05  @app.route("/", methods=['GET','POST'])
06  def index():
07      if request.method == "GET":
08          return render_template("results.html")
09      else:
10          name = request.form["name"]
11          score = int(request.form["score"])
12          return render_template("results.html", name=name, score=score)
13
14  if __name__ == '__main__':
        app.run()
```

results.html

```
01  <!DOCTYPE HTML>
02
03  <html>
04        <head>
05              <title>Results</title>
06        </head>
07
08        <body>
09              <h1>Results</h1>
10
11              <!-- input your form here -->
12
13              <br><br>
14
15              <!-- input your conditional statements here -->
16
17        </body>
18  </html>
```

Complete results.html template which takes in a name and a score using a form. Browser should display “Hello <name>. Your score is <score>. You passed.” if score is more than or equals to 50, else browser should display. “Hello <name>. Your score is <score>. You failed.”

5.2 For statements

In Jinja, a for statement is used to loop over an iterable object, such as a list, tuple, string or a dictionary, and perform a certain action for each item in the iterable.

A typical use of for loop is to output the contents of a collection in a table:

table.py

```
01 from flask import Flask, render_template
02
03 app = Flask(__name__)
04
05 @app.route('/')
06 def home():
07     results={
08         'English':75,
09         'Mathematics':73,
10         'Mother Tongue':60,
11         'Science':75
12     }
13     return render_template('table.html', results=results)
14
15 if __name__ == '__main__':
16     app.run()
```

table.html

```
01 <!DOCTYPE HTML>
02
03 <html>
04   <head>
05     <title>Table of Results</title>
06   </head>
07
08   <body>
09     <h1>Table of Results</h1>
10
11     <table>
12       <tr>
13         <th>Subject Name</th>
14         <th>Score</th>
15       </tr>
16       {% for subject in results %}
17       <tr>
18         <td>{{ subject }}</td>
19         <td>{{ results[subject] }}</td>
20       </tr>
21       {% endfor %}
22     </table>
23
24   </body>
25 </html>
```

The home function renders a template named table.html using render_template() function and passes the results dictionary as a parameter. The table.html template can

access the results dictionary using the variable results in its Jinja code and can display the marks for each subject in a table format.

Exercise 5

- Complete the get_grade() function that accepts an integer score and returns its corresponding grade.
- Process the scores from user input and parse the results dictionary containing the subject with corresponding grades as an argument to table_grade.html template.
- Complete table_grade.html by adding a form for users to enter scores and a table to show the subjects and their corresponding grades.

table_grade.py

```
01 from flask import Flask, render_template, request
02
03 app = Flask(__name__)
04
05 def get_grade(score):
06
07     grade = ''
08     # write your code here
09     return grade
10
11 @app.route("/", methods=['GET','POST'])
12 def home():
13     if request.method == "GET":
14         return render_template("table_grade.html")
15
16     results={
17         'GP': '',
18         'Mathematics': '',
19         'Computing': '',
20         'Econs': ''
21     }
22
23     # write your code here
24
25     return render_template("table_grade.html", results=results)
26
27 if __name__ == '__main__':
28     app.run()
29
```

results.html

```
01 <!DOCTYPE HTML>
02
03 <html>
04   <head>
05     <title>Table of Results</title>
06   </head>
07   <body>
08     <h1>Table of Results</h1>
09
10     <!-- insert your form here -->
11
12     <br><br>
13
14     <!-- insert your code for display of results here-->
15
16   </body>
17 </html>
```

6 External Cascading Style Sheet

Let's recall a typical file hierarchy of a Flask project:

1. **The main directory**: This directory contains the main application file and other directories, including app.py, the main Flask application file that defines the routes, views, models, and other components of the application.
2. The **templates** directory: This directory contains the HTML templates used for rendering the views of the application.
3. The **static** directory: This directory contains static files, including CSS, JavaScript, and image files, used for styling and functionality in the application.

In Flask projects, external CSS files are typically stored in the "static" directory, and their reference links are included in the <head> section of HTML templates using the <link> tag:

```
<head>
      <link rel="stylesheet" href="/static/style.css">
</head>
```

To prevent hardcoding the path to external stylesheets in Flask projects, the url_for() function can be used to access the stylesheet files. For instance:

```
<head>
  <link rel="stylesheet" href="{{ url_for('static',filename='styles.css') }}">
</head>
```

This code generates a URL for the "styles.css" file in the "static" directory, based on the Flask application's configuration.

Exercise 6

- Create a static folder. Create a style.css file and save it in the static folder.
- Add this statement into <head> of html:
  <link rel="stylesheet" href="{{ url_for('static',filename='style.css') }}">
- Add a 1px solid black styling to the table of grades by subject.

7 Handling file and image upload

When an <input> tag has its type attribute set to "file", this allows the user to upload files for submission with the form. However, for file uploads to work properly, the enclosing <form> must also be configured to use POST and include an additional enctype attribute that is set to "multipart/form-data".

When set to "multipart/form-data", it indicates that the form data will be divided into multiple parts, each part containing a portion of the data along with its corresponding content type, and then sent to the server. This is typically used when the form includes a file input field, as files cannot be represented as simple key-value pairs in the form data.

In Flask, when a form is submitted with enctype="multipart/form-data", the form data can be accessed using the request.files attribute, which allows you to handle files and other form data that has been sent as part of the form submission.

Create another folder uploads alongside templates and static such that files uploaded will be stored in the uploads folder.

Upload_file.py

```
01  import os
02  from flask import Flask, render_template, request, send_from_directory
03
04  app = Flask(__name__)
05
06  photolist = []
07
08  @app.route('/', methods=['GET', 'POST'])
09  def index():
10      if request.method == 'GET':
11          return render_template("index.html")
12
13      #post
14      image = request.files['photo']
15      path = os.path.join('uploads', image.filename) # save the photo to
16  uploads folder
17      image.save(path)
18
19      photolist.append(image.filename)
20      return render_template('index.html', filename = image.filename,
21  images = photolist)
22
23  @app.route('/photos/<filename>')
24  def get_file(filename):
25      return send_from_directory('uploads', filename)
26
27  app.run()
```

Note that unlike normal form data, submitted files are not accessed from request.form but from a separate request.files dictionary. Each value in this dictionary is a file object with a filename attribute as well as a save() method that accepts a file path and writes the submitted file onto the server's file system using the provided file path.

Inside the get_file function, the send_from_directory function is called with two arguments: the first argument is the directory from which the file should be served, and the second argument is the name of the file that should be sent to the client.

In this specific case, the send_from_directory function sends a file from the 'uploads' directory with the filename that matches the filename variable from the URL. This allows Flask to serve uploaded files to clients who request them.

upload_file.html

```
<!DOCTYPE html>

<html>
  <body>
    <h1>File upload</h1>

    <form method = "post" enctype="multipart/form-data">
      <input type="file" name="photo" />
      <input type="submit" />
    </form>
    {% for image in images %}
      <img src="photos/{{ image }}" />
    {% endfor %}
  </body>
</html>


```

Each <img> tag will cause the client's browser to send a request to the Flask application for the corresponding image file. The Flask application, in turn, will use the send_from_directory function to retrieve the file from the uploads directory and send it back to the client as a response.