Web Applications

Chapter No.	33
Status	Completed

- Comparing Native Applications & Web Applications
  - Native Applications
    - A native application is an application software that is developed for use on a particular platform or device
    - Advantages
      - Native applications are usually more efficient than web applications because native applications are customised for a specific platform
      - Native applications tend to have more direct access to device features, such as the camera, GPS and accelerometer, as compared to web applicaitons
      - Native applications can function without Internet access, unlike web applications
    - Disadvantages
      - It is more costly to develop native applications because every platform and operating system requires a different version of the native application, unlike web applications, which can run on any device that can run a web browser
      - Native applications require users to install updates, either manually or automatically, which creates more hassle as compared to web applications, which do not require its users to install updates
      - Users have to download and install native applications in order to use them, which creates more hassle as compared to web applications, which do not require any download or installation to be used
  - Web Applications
    - A web application is an application software that runs on a web server and is accessed by a web browser
    - Advantages
      - Web applications are more platform independent than native applications because web applications can run on any platform that can run a web browser, unlike native applications, which require a specific version of the native application to be developed for each platform
      - It is easier to maintain a web application with a single code base as compared to a native application with multiple code bases for different versions for different platforms
      - Users do not need to install updates to use web applications because the updating is done on the server-side, unlike native applications, which require users to install updates
      - Users do not need to download and install web applications to use them, unlike native applications, which require users to download and install them
      - Web applications are more easily shared as compared to native applications as users only need to share a URL in order to share a web application, while users have to share the program files of a native application in order to share it

- ▼ Disadvantages
  - Web applications usually have less access to device features, such as the camera, GPS and accelerometer, as compared to native applications
  - Web applications may not function in the same way across different browsers or even different versions of the same browser
  - Web applications require Internet access in order to be used, unlike native applications
  - Web applications may have a less ideal user experience as compared to native applications because web applications are not customised for any specific platform, while native applications are specifically designed for a certain platform or operating system

- ▼ Serving Webpages with Flask & Jinja2
  - ▼ Flask is a Python framework that allows us to use Python to serve up web pages
    - ▼ @app.route("/pathname")
      - Decorator that associates a Python function with a path
    - ▼ app.run( )
      - Runs the web application
    - ▼ app.run(debug=True)
      - Runs the web application and causes errors in the web application to be displayed on the webpage
    - ▼ request.args
      - A dictionary-like object that contains data submitted through the form when the GET method is used
      - The names of the input fields act as dictionary keys
    - ▼ request.form
      - A dictionary-like object that contains data submitted through the form when the POST method is used
      - The names of the input fields act as dictionary keys
    - ▼ request.files
      - A dictionary-like object that contains files submitted through the form
      - The names of the input fields act as dictionary keys
  - ▼ HTML and CSS is used to format and beautify the page, while Python operates in the background to process user requests and user data
    - HTML files to be displayed by the web application are placed in the templates folder
    - ▼ <form> Attributes
      - ▼ action
        - Determines where the form data is submitted to
      - ▼ method
        - Specifies the http method to be used ("GET" or "POST") when the form is submitted
        - Defaults to "GET" if the method attribute is omitted

▼ Jinja2 is a web template language that works in conjunction with Flask

▼ {{ }} represents 'print' in Jinja2

- {{name}} in the HTML file will print out the value of the name variable

▼ {{url_for("display")}} in the HTML file instructs Python to retrieve the web address associated with the display function in the Python file

- Hence, when the form on the webpage is submitted, Python will look up the display function and execute the function to serve up a webpage

▼ GET & POST

- Sending data using the GET method appends the data being sent to the URL, making it less secure
- Sending data using the POST method does not cause the data being sent to appear in the URL, making it more secure
- Flask defaults to using the GET method
- In order to use the POST method, it must be specified in both the HTML form attribute and the Python function

▼ Example - Basic Web Application

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
    <title>Form Page</title>
  </head>
  <body>
    <form action="{{url_for('display')}}" method="POST">
      First Name: <input type="text" name=firstname><br>
      Last Name: <input type="text" name=lastname><br>
      Age: <input type="text", name=age><br>
      <input type="submit" value="Click here to submit!">
    </form>
  </body>
</html>
```

```
#server.py
from flask import Flask, render_template, request

app = Flask(__name__) #creates a Flask object

@app.route("/") #associates the path / with the function below
def root(): #this function gets executed whenever we visit the path /
  return render_template("index.html") #displays index.html

@app.route("/display", methods=["POST"]) #associates the path /display with the function below
def display(): #this function gets executed whenver we visit the path /display
  firstname = request.form["firstname"]
  lastname = request.form["lastname"]
  age = request.form["age"]
  return render_template("display.html", firstname=firstname, lastname=lastname, age=age)

app.run()
```

```
<!-- display.html -->
<!DOCTYPE html>
<html>
  <head>
    <title>Display Page</title>
  </head>
```

```
  <body>
    <p>Your name is: {{firstname}} {{lastname}}</p>
    <p>Your age is: {{age}}</p>
  </body>
</html>
```

▼ FOR Loops in Jinja2

```
{% for item in data % }
...
{% endfor %}
```

▼ Conditionals in Jinja2

```
{% if firstname|length > 5 %}
...
{% elif surname|length < 3 %}
...
{% else %}
...
{% endif %}
```

- The Jinja2 |length filter obtains the length of a string

▼ Displaying HTML Code with Jinja2

```
#server.py
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def root()
  html_code = "<b>This is bolded</b>"
  return render_template("index.html", html_code=html_code)

app.run()
```

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    This is a sample of html code: {{html_code|safe}}
  </body>
</html>
```

- The Jinja2 |safe filter allows Jinja2 to directly display html code obtained from other sources
- If the |safe filter isn't used, Jinja2 will display the html code like normal text

▼ Redirecting with Flask

- There are times where you might want to redirect a user from an outdated webpage to an updated webpage

```
from flask import Flask, render_template, redirect, url_for

app = Flask(__name__)

@app.route("/")
def root_old():
  return redirect(url_for('root_new'))

@app.route("/new")
def root_new():
  return render_template("index.html")

app.run()
```

▼ CSS with Flask

- As CSS files are considered static files, they need to be placed in the static folder
- By default, Flask looks for static files in the static folder

```
<!--index.html-->
<!DOCTYPE html>
<html>
  <head>
    <title>This is a title</title>
    <link rel="stylesheet" href="{{url_for('static', filename='style.css')}}">
  </head>
  <body>
    <p id="p1">Text 1</p>
    <p id="p2">Text 2</p>
  </body>
</html>
```

```
/* style.css */

#p1 {
    color:red;
}

#p2 {
    color:blue;
}
```

▼ Handling File Uploads with Flask

- POST method must be used

▼ secure_filename( )

- Secures a filename to prevent malicious filename from interfering with the web application

▼ os.path.join( )

- Joins two paths together

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
```

```
    <title>Form Page</head>
  </head>
  <body>
    <form action="{{url_for('display')}}" method="POST" enctype="multipart/form-data">
      File Upload: <input type="file" name="file"><br>
      Image Upload: <input type="file" name="image"><br>
      <input type="submit">
    </form>
  </body>
</html>
```

```
#server.py
from flask import Flask, render_template, request
import os
from werkzeug.utils import secure_filename

app = Flask(__name__)

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

@app.route("/display", methods=["POST"])
def display():
  f1 = request.files["file"]
  line1 = f1.readline().decode() #decodes the byte string into a normal string

  image = request.files["image"]
  image_path = os.path.join("static", secure_filename(image.filename)) #.filename is an attribute of a file object
  image.save(image_path) #saves the file at the specified path

  return render_template("display.html", line1=line1, image=image.filename)

app.run()
```

```
<!-- display.html -->
<!DOCTYPE html>
<html>
  <head>
    <title>Display Page</title>
  </head>
  <body>
    First Line in the Uploaded File: {{line1}}<br>
    Uploaded Image: <img src="{{url_for('static', filename=image)}}">
  </body>
</html>
```

▼ Integrating SQL into Web Applications

- Essentially integrating database functions into a web application using the sqlite3 module

▼ Example - Data Entry Web Application

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
    <title>Form Page</title>
  </head>
  <body>
    <form action="{{url_for('display')}}">
      Name: <input type="text" name="name"><br>
      Class: <input type="text" name="class_"><br>
      Age: <input type="text" name="age"><br>
      <input type="submit">
    </form>
```

```
  </body>
</html>
```

```
#server.py
from flask import Flask, render_template, request
import sqlite3

app = Flask(__name__)

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

@app.route("/display")
def display():
  name = request.args["name"]
  class_ = request.args["class_"]
  age = int(request.args["age"])

  connection = sqlite3.connect("acjc.db")
  cursor = connection.cursor()

  cursor.execute('''
                 CREATE TABLE IF NOT EXISTS Student(
                 Name TEXT PRIMARY KEY,
                 Class TEXT,
                 Age INTEGER)
                 ''')

  cursor.execute('''
                 INSERT INTO Student
                 VALUES(?, ?, ?)
                 ''', (name, class_, age))

  connection.commit()
  connection.close()

  return render_template("display.html", name=name, class_=class_, age=age)

app.run()
```

```
<!-- display.html -->
<!DOCTYPE html>
<html>
  <head>
    <title>Display Page</title>
  </head>
  <body>
    The following information has been stored:<br>
    Name: {{name}}<br>
    Class: {{class_}}<br>
    Age: {{age}}<br>
  </body>
</html>
```

▼ Variable Routes in Flask

- Adding <s> to the route allows a variable to be passed to the function
- This variable can then be worked on by Python or Jinja2
- If the variable route is an empty string, the webpage does not work as intended
- ▼ You can restrict the allowable inputs to integers only by specifying <int:s>
  - If non-integers are specified, a 404 error is returned

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    You entered the number {{num}}!
  </body>
</html>
```

```
#server.py
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/<int:s>")
def root(s):
    return render_template("index.html", num=s)

app.run()
```

▼ Multiple Routes in Flask

- You can decorate a function with more than one route
- The same webpage can be reached with multiple paths

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <h1>This is my webpage</h1>
  </body>
</html>
```

```
#server.py
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
@app.route("/apple")
def root():
  return render_template("index.html")

app.run()
```

▼ Usability Principles of Web Applications

▼ Principle 1 - Keep users informed of the system's status

- Download status bar
- Wi-Fi icon
- Battery level indicator

▼ Principle 2 - Match between system and the real world

- Use phrases, icons and concepts understandable by the user
- E-book readers allow users to turn the page by swiping the screen from right to left (corresponding to how you flip a physical book)
- Volume control buttons (top button increases volume, bottom button decreaes volume)

▼ Principle 3 - User control and freedom

- Allow users to undo or redo
- Allow users to return to the previous menu
- Allow users to exit the app easily

▼ Principle 4 - Consistency and standards (internal consistency and external consistency)

- Follow conventions and use the same term to mean the same thing consistently
- Users should not have to guess if two different terms are referring to the same thing
- 'Close window' button is always at the top right corner in Windows
- Almost all online shopping websites have a shopping cart page and use a shopping cart icon as it is what people expect to see

▼ Principle 5 - Error prevention

- Include helpful constraints (using a drop down list or radio button to restrict users to only valid choices)
- Confirmation dialogue box
- Undo button for people to prevent them from making permanent errors

▼ Principle 6 - Recognition rather than recall

- Make objects and options clearly visible
- Use common icons
- Shopping websites provide "previously bought items" for users to recognise what they last bought (and probably want to buy again)
- Microsoft Word shows a list of recently opened documents and common templates

▼ Principle 7 - Flexibility and efficiency of use

- Allow multiple ways of achieving the same result

▼ Principle 8 - Aesthetic and minimalist design

- Provide only what is necessary (redundant information clutters the screen and competes with useful information)

▼ Principle 9 - Help users recognise, diagnose and recover from errors

- Provide error messages in simple and language
- Be specific with error messages to allow the user to pinpoint the exact cause of the error and rectify it

▼ Principle 10 - Help and documentation

- Ideally, an app should be usable without any documentation
- It is still a good idea to provide some documentation to assist users

- Documentation should be as concise as possible