Python and Flask

COMP1048: Databases and Interfaces (2024-2025)

Matthew Pike & Yuan Yao

Table of contents

1 Lab Overview

In this lab, you will begin using Python and the Flask framework. This lab is designed to introduce you to the fundamentals of building a web application, where you'll learn:

- **Developing a Basic Flask App**
- **Generating Dynamic Content**
- **Using Templates**
- **Integrating Style Sheets**

By the end of this lab, you will have developed essential skills in web development with Python and Flask, laying a solid foundation for Coursework 2.

2 Getting Started

This section will walk you through the basic steps to get your development environment ready for the tasks ahead.

Step 1: Open Your Code Editor

Begin by launching your preferred code editor. Visual Studio Code is recommended as it supports Python and web technologies (HTML, CSS).

Step 2: Set Up Your Project Directory

Create a new directory (folder) on your computer, naming it with your student ID, for example, 12345678. This will be your main project directory. Within this folder, create two additional subdirectories: `static` and `templates`.

Step 3: Create Your Main Python File

Now, within your project directory (your student id), create a new file named `app.py`. Your project directory should now look like this:

```
<student-id>/      # Root directory named after your student ID
├── templates/     # Directory for your HTML templates
└── static/        # Directory for static files
└── app.py         # Your main Flask Python file
```

`app.py` will contain the basic boilerplate code for your Flask application. Start with the following basic structure:

```python
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return ""

if __name__ == "__main__":
   app.run(debug=True, port=5000)
```

ℹ ModuleNotFoundError

If when running your application you receive the error message `ModuleNotFoundError: No module named 'flask'`, this means you need to install the Flask module. Please refer to Lab 001 for instructions on how to install Flask.

Step 4: Open a CLI

Open a CLI (Mac: Terminal, Windows: Command Prompt) and navigate to your project directory (your student id). If you're unsure how to do this, refer to Lab 001 for detailed instructions. Make sure you're in the correct directory by running the `ls` command (Mac) or `dir` command (Windows) - you should see the `app.py` file you created in Step 3.

Step 5: Run Your Application

In the CLI, run your Flask application by typing `python app.py` or `python3 app.py`. This should start your Flask server.

Step 6: Open Your Web Browser

Open your web browser and navigate to `http://127.0.0.1:5000`. You should see be greeted with a blank white page, but no errors.

Step 7: Check Terminal Output

Lastly, observe the output in your CLI. It should show logs indicating server activity, such as requests being made when you access the URL in your browser. Try refreshing the page in your browser and observe the CLI output again.

Step 8: Moving Forward

If you want to stop your Flask application, press `Ctrl+C` in the CLI. Note that because we've enabled debug mode, the server will automatically restart when you make changes to your code, so you do not need to manually restart the server each time you make a change.

You are now ready to begin the tasks below.

3 Tasks

3.1 Hello World!

In this task, you modify your `app.py` file containing your web application to display "Hello World!" in a web browser. When a user visits the root URL (/), the application should directly show "Hello World!". There is no need to use templates, HTML, or CSS for this task; simply return the string "Hello World!". To view the output, users will need to visit `http://127.0.0.1:5000` in their web browser.

3.2 Hello <name>!

In this task, you are to enhance your application to display a greeting that includes a user-specified name. The message "Hello `<name>`!" should be displayed, where `<name>` is a variable obtained from the URL. To implement this, set up a route `/hello/<name>` in your application. For example, if a user visits `http://127.0.0.1:5000/hello/Alice`, the webpage should show "Hello Alice!". Like the previous task, there is no requirement to use templates, HTML, or CSS. Your goal is to simply return the string "Hello `<name>`!", with `<name>` dynamically replaced by the name provided in the URL.

Function Naming

If you encounter an `AssertionError: View function mapping is overwriting an existing endpoint function`, it usually indicates a naming conflict. This can happen if you've named your new function `hello`, which conflicts with the existing root URL (/). To resolve this, rename your function to something distinct, such as `hello_name`.

3.3 Hello <name>! (with templates)

Extend your web application to generate HTML content using a template. It's important to maintain the existing functionality of `/hello/<name>` as established in Section 3.2. To accomplish this, follow these steps:

1. Create a template file named `hello.html` within the `templates` folder. In this template, utilise the `{{ name }}` variable to dynamically display the user's name. Enclose the variable within `<h1>` tags to present the name prominently as a heading.
2. Import the `render_template` function from the `flask` module. This function is essential for rendering your HTML template.
3. Modify the `hello_name` function to incorporate the `render_template` function. Make sure to pass the `name` variable to the template as an argument.

Common Issues

There are two common issues that students encounter when attempting this task:

1. **`TemplateNotFound` error:** This error indicates that the template file could not be found. Ensure that your `hello.html` template is correctly placed in the `templates` folder. If you encounter issues, double-check this common setup mistake.

2. `NameError: name 'render_template' is not defined` **error:** This error indicates that the `render_template` function could not be found. Ensure that you've imported the `render_template` function from the `flask` module at the top of your Python file.

3.4 Adding Style

Create a basic CSS stylesheet to enhance the visual appearance of your web application. Begin by creating a file named `style.css` within the `static` folder of your project. In this file, add some fundamental CSS rules. For example, you could change the background colour of the page. The `style.css` file will be utilised in the subsequent task.

Remember, the focus of this lab is on Flask and Python, so this step is just to introduce a basic level of styling to your application.

3.5 Template Inheritance

Next, you'll be introducing the concept of template inheritance to your web application. Start by creating a base template named `base.html` in the `templates` folder. This base template should include the essential structure for your web pages:

- Begin with the `<!DOCTYPE html>` declaration and the root `<html>` element.
- Include a `<head>` element, within which you should place a `<title>` element. Inside the `<title>`, use `{% block title %}{% endblock %}`. This allows you to set the title of each page individually.
- Also within the `<head>` element, link the CSS file you created in Section 3.4. Use the `url_for` function to link the CSS file. This function is crucial for generating URLs for static files, like your CSS file. Remember that you need to import the `url_for` function from the `flask` module. For example:
  - `<link rel="stylesheet" href="{{{url_for('static',filename='style.css')}}}">`
- Add a `<body>` element that contains a `<div>` element. Inside this `<div>`, include `{% block content %}{% endblock %}`. This setup allows for individual page content to be defined separately in different templates.

You now have a basic template that can be extended by other templates, something you'll be doing in the next task.

3.6 Adding a Form

Modify your web application to incorporate a form. Create a new template named `message.html` within the `templates` folder. The new template should extend the `base.html` template you created in the previous task. In the `content` block, add a `<form>` element, followed by these form elements:

- Text input field: User's name (`name="username"`)
- Text input field: User's email address (`name="email"`)
- Text input field: User's message (`name="message"`)
- Submit button

The form's `action` attribute should be set to `/message/`, and the `method` attribute should be set to `POST`. The `action` attribute specifies the URL to which the form data will be submitted, and the `method` attribute specifies the HTTP method to use when submitting the form data.

3.7 Handling Form Submission

In this final task, you will extend your web application to handle form submissions. The goal is to display the user's name, email address, and message on the webpage upon form submission. To achieve this, follow these steps:

1. Create a new function named `message` in your Flask application. This function should be responsible for handling the `/message/` URL. Ensure it renders the `message.html` template you previously created, when a GET request is made to the `/message/` URL.

Checking for GET Requests

You can use `request.method` to check the type of request being made, for example:

```
if request.method == "GET":
    # Handle GET request
```

Remember to import the `request` object from the `flask` module.

2. Within the `message` function, implement code to retrieve form data using Flask's request handling.
   You'll need to extract data such as the user's name, email address, and message from the submitted form. For example, to retrieve the user's email, you could use `request.form.get("email")`.
3. Modify the `message.html` template to display the retrieved form data. Use placeholders within the template (like `{{ name }}`, `{{ email }}`, and `{{ message }}`) to show the user's details.
4. Ensure that your application is set up to handle both `GET` and `POST` requests at the `/message/` URL.
   This allows the form to be displayed initially (GET request) and then process the form data upon submission (POST request).

Enabling GET and POST Requests

To enable both GET and POST requests, you'll need to modify the `methods` argument of the `@app.route` decorator. For example, to enable both GET and POST requests, you could use:

```
@app.route("/message/", methods=["GET", "POST"])
```

5. Test your form submission by running the application, filling in the form, and verifying that the user's details are correctly displayed on the webpage after submission.

By completing this task, you will have successfully added interactive functionality to your web application, allowing it to handle and display user-provided information.

3.8 Bonus Task: Using the `flash` Function

While it is not a requirement for the lab, completing this task will deepen your understanding of Flask and enhance your web application.

In this task, you will use Flask's `flash` function to display temporary messages to the user. These messages are useful for showing feedback, such as success or error messages, after certain actions (like form submissions).

To complete this task:

1. **Implement Flash Messaging**: Modify your `message` function to flash a message if there are any errors in the form submission. For example, if the user fails to provide their name, email address, or message, you might want to flash an error message - "Please fill in all fields".

Logical Operators in Python

You can use logical operators to check if a variable is empty, for example:

```python
if not name or not email or not message:
  # Name, email, or message is empty
...

if name and email and message:
  # Name, email, and message are not empty
```

2. **Configure the Base Template**: Update your `base.html` template to display flash messages. You can use the following code to display flash messages:

```html
{% with messages = get_flashed_messages() %}
  {% if messages %}
    {% for message in messages %}
      <div class="alert">
        {{ message }}
      </div>
    {% endfor %}
  {% endif %}
{% endwith %}
```

4 Submitting Your Lab Work

Please submit a ZIP file named with your student ID (e.g., `12345678.zip`) containing the assets of your Flask web application. Ensure your submission follows this structure:

```
<student-id>/       # Root directory named after your student ID
├── app.py          # Your main Flask Python file
├── templates/        # Directory for your HTML templates
│   ├── base.html     # Your base template file
│   ├── hello.html    # Template for the Hello World task
│   └── message.html  # Template for displaying form submissions
├── static/           # Directory for static files
│   └── style.css     # Your CSS file
```

Submit your work to the Lab 9 assignment on Moodle before the deadline. Remember that late submissions will be penalised with a mark of zero, as outlined in the coursework issue sheet.

Your submission must be legible, uncorrupted, and demonstrate a sincere effort in completing the lab tasks. Submissions that fail to meet these criteria will be given a mark of zero.

This lab constitutes 2% of your total grade for the module.