MASTER OF COMPUTER APPLICATIONS

SEMESTER 1

PYTHON PROGRAMMING

Unit 11

Python for Web Development

Table of Contents

1. INTRODUCTION

Welcome to Unit 11 of our programming course, where we dive into the dynamic world of Python for Web Development. In Unit 10, you've mastered the fundamentals of database interaction, exploring SQL basics, the practicalities of the SQLite module, essential CRUD operations, and the critical concept of transactions. This solid foundation in managing and manipulating data is crucial as we move forward, given that web development heavily relies on data exchange and manipulation.

Unit 11 shifts our focus from backend storage to how we interact with users through the web. Python, with its simplicity and versatility, is a powerful tool for web development. In this unit, we will introduce Flask, a lightweight and powerful web framework that makes it easy to build simple, yet robust web applications. Flask uses Python's straightforward syntax to extend web applications with functionalities like user authentication, blogging capabilities, and interactive features, making it an excellent choice for both beginners and seasoned developers looking to create scalable web applications.

To navigate this unit effectively, we'll start by setting up a proper development environment, utilizing virtual environments to manage dependencies. You'll learn how to structure a Flask application, manage request-response cycles, and utilize templates to render dynamic content. Each concept will be reinforced with hands-on examples, including building a social blogging application from scratch. By the end of this unit, you'll not only understand the technical aspects of web development with Flask but also appreciate how these tools can be used to solve real-world problems efficiently. So, let's begin our journey into the exciting world of web development with Flask, enhancing your skillset and preparing you for advanced web development projects.

1.1 Learning Objectives

At the end of this unit, you will be able to:

- ❖ *Identify the components required to set up a Flask application, including virtual environments and Flask-specific configurations.*
- ❖ *Explain the function and purpose of routing in Flask and how routes connect URLs to Python view functions.*
- ❖ *Construct a simple Flask application that demonstrates basic routing and template use.*
- ❖ *Differentiate between GET and POST methods in web form submissions and their handling in Flask.*
- ❖ *Assess the use of Flask extensions in enhancing application functionality, such as Flask-Login for user authentication and Flask-SQLAlchemy for database interactions.*
- ❖ *Design a web application using Flask that incorporates user authentication, database interactions, and dynamic content generation through Jinja2 templates.*

2. INTRODUCTION TO FLASK

Flask is a small framework by most standards—small enough to be called a "micro-framework," and small enough that once you become familiar with it, you will likely be able to read and understand all of its source code.

But being small does not mean that it does less than other frameworks. Flask was designed as an extensible framework from the ground up; it provides a solid core with the basic services, while extensions provide the rest. Because you can pick and choose the extension packages that you want, you end up with a lean stack that has no bloat and does exactly what you need.

Flask has three main dependencies. The routing, debugging, and Web Server Gateway Interface (WSGI) subsystems come from Werkzeug; the template support is provided by Jinja2; and the command-line integration comes from Click. These dependencies are all authored by Armin Ronacher, the author of Flask.

Flask has no native support for accessing databases, validating web forms, authenti- cating users, or other high-level tasks. These and many other key services most web applications need are available through extensions that integrate with the core pack- ages. As a developer, you have the power to cherry-pick the extensions that work best for your project, or even write your own if you feel inclined to. This is in contrast with a larger framework, where most choices have been made for you and are hard or sometimes impossible to change.

2.1 Creating the Application Directory

To begin, you need to create the directory that will host the example code, which is available in a GitHub repository. The most convenient way to do this is by checking out the code directly from GitHub using a Git client. The following commands download the example code from GitHub and initialize the application to version 1a, which is the initial version you will work with:

```
$ git clone https://github.com/miguelgrinberg/flasky.git

$ cd flasky

$ git checkout 1a
```

2.2 Virtual Environments

Now that you have the application directory created, it is time to install Flask. The most convenient way to do that is to use a virtual environment. A virtual environment is a copy of the Python interpreter into which you can install packages privately, without affecting the global Python interpreter installed in your system.

Virtual environments are very useful because they prevent package clutter and version conflicts in the system's Python interpreter. Creating a virtual environment for each project ensures that applications have access only to the packages that they use, while the global interpreter remains neat and clean and serves only as a source from which more virtual environments can be created. As an added benefit, unlike the system-wide Python interpreter, virtual environments can be created and managed without administrator rights.

2.3 Creating a Virtual Environment with Python 3

The creation of virtual environments is an area where Python 3 and Python 2 interpreters differ. With Python 3, virtual environments are supported natively by the venv package that is part of the Python standard library.

The command that creates a virtual environment has the following structure:

```
$ python3 -m venv virtual-environment-name
```

The -m venv option runs the venv package from the standard library as a standalone script, passing the desired name as an argument.

You are now going to create a virtual environment inside the flasky directory. A commonly used convention for virtual environments is to call them venv, but you can use a different name if you prefer. Make sure your current directory is set to flasky, and then run this command:

```
$ python3 -m venv venv
```

After the command completes, you will have a subdirectory with the name venv inside flasky, with a brand-new virtual environment that contains a Python inter- preter for exclusive use by this project.

2.4 Working with a Virtual Environment

When you want to start using a virtual environment, you have to "activate" it. If you are using a Linux or macOS computer, you can activate the virtual environment with this command:

```
$ source venv/bin/activate
```

If you are using Microsoft Windows, the activation command is:

```
$ venv\Scripts\activate
```

When a virtual environment is activated, the location of its Python interpreter is added to the PATH environment variable in your current command session, which determines where to look for executable files. To remind you that you have activated a virtual environment, the activation command modifies your command prompt to include the name of the environment:

```
(venv) $
```

After a virtual environment is activated, typing python at the command prompt will invoke the interpreter from the virtual environment instead of the system-wide interpreter. If you are using more than one command prompt window, you have to activate the virtual environment in each of them.

2.5 Installing Python Packages with pip

Python packages are installed with the pip package manager, which is included in all virtual environments. Like the python command, typing pip in a command prompt session will invoke the version of this tool that belongs to the activated virtual environment.

To install Flask into the virtual environment, make sure the venv virtual environment is activated, and then run the following command:

```
(venv) $ pip install flask
```

When you execute this command, pip will not only install Flask, but also all of its dependencies. You can check what packages are installed in the virtual environment at any time using the pip freeze command:

```
(venv) $ pip freeze

click==6.7 Flask==0.12.2

itsdangerous==0.24

Jinja2==2.9.6

MarkupSafe==1.0

Werkzeug==0.12.2
```

The output of pip freeze includes detailed version numbers for each installed pack- age. The version numbers that you get are likely going to be different from the ones shown here.

You can also verify that Flask was correctly installed by starting the Python inter- preter and trying to import it:

```
(venv) $ python

>>> import flask

>>>
```

If no errors appear, you can congratulate yourself: you are ready for the next chapter, where you will write your first web application.

3. PARTS OF A FLASK APPLICATION

3.1 Initialization

All Flask applications must create an application instance. The web server passes all requests it receives from clients to this object for handling, using a protocol called Web Server Gateway Interface (WSGI, pronounced "wiz-ghee"). The application instance is an object of class Flask, usually created as follows:

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

The only required argument to the Flask class constructor is the name of the main module or package of the application. For most applications, Python's name variable is the correct value for this argument.

3.2 Routes and View Functions

Clients such as web browsers send requests to the web server, which in turn sends them to the Flask application instance. The Flask application instance needs to know what code it needs to run for each URL requested, so it keeps a mapping of URLs to Python functions. The association between a URL and the function that handles it is called a route.

The most convenient way to define a route in a Flask application is through the app.route decorator exposed by the application instance. The following example shows how a route is declared using this decorator:

```
@app.route('/')

def index():

    return '<h1>Hello World!</h1>'
```

The previous example registers function index() as the handler for the application's root URL. While the app.route decorator is the preferred method to register view functions, Flask also offers a more traditional way to set up the application routes with the app.add_url_rule() method, which in its most basic form takes three arguments: the URL, the endpoint name, and the view function. The following example uses app.add_url_rule() to register an index() function that is equivalent to the one shown previously:

```
def index():
    return '<h1>Hello World!</h1>'
app.add_url_rule('/', 'index', index)
```

Functions like index() that handle application URLs are called view functions. If the application is deployed on a server associated with the www.example.com domain name, then navigating to http://www.example.com/ in your browser would trigger index() to run on the server. The return value of this view function is the response the client receives. If the client is a web browser, this response is the document that is displayed to the user in the browser window. A response returned by a view function can be a simple string with HTML content, but it can also take more complex forms, as you will see later.

If you pay attention to how some URLs for services that you use every day are formed, you will notice that many have variable sections. For example, the URL for your Facebook profile page has the format https://www.facebook.com/<your-name>, which includes your username, making it different for each user. Flask supports these types of URLs using a special syntax in the app.route decorator. The following example defines a route that has a dynamic component:

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {}!</h1>'.format(name)
```

The portion of the route URL enclosed in angle brackets is the dynamic part. Any URLs that match the static portions will be mapped to this route, and when the view function is invoked, the dynamic component will be passed as an argument. In the preceding example, the name argument is used to generate a response that includes a personalized greeting.

The dynamic components in routes are strings by default but can also be of different types. For example, the route /user/<int:id> would match only URLs that have an integer in the id dynamic segment, such as /user/123. Flask supports the types string, int, float, and path for routes. The path type is a special string type that can include forward slashes, unlike the string type.

3.3 A Complete Application

In the previous sections you learned about the different parts of a Flask web application, and now it is time to write your first one. The hello.py application script shown in Example 1 defines an application instance and a single route and view function, as described earlier.

Example 1: hello.py: A complete Flask application

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

@app.route('/')

def index():

    return '<h1>Hello World!</h1>'
```

3.4 Development Web Server

Flask applications include a development web server that can be started with the flask run command. This command looks for the name of the Python script that contains the application instance in the FLASK_APP environment variable.

To start the hello.py application from the previous section, first make sure the virtual environment you created earlier is activated and has Flask installed in it. For Linux and macOS users, start the web server as follows:

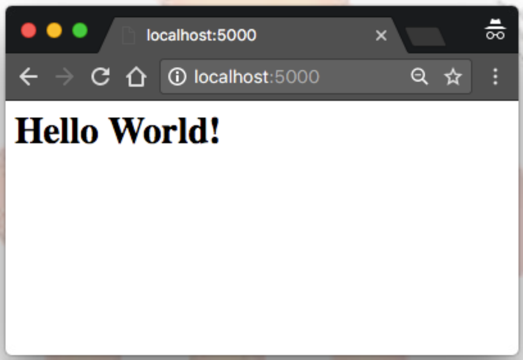
```
(venv) $ export FLASK_APP=hello.py

(venv) $ flask run

* Serving Flask app "hello"

* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

For Microsoft Windows users, the only difference is in how the FLASK_APP environment variable is set:

```
(venv) $ set FLASK_APP=hello.py

(venv) $ flask run

* Serving Flask app "hello"
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Once the server starts up, it goes into a loop that accepts requests and services them. This loop continues until you stop the application by pressing Ctrl+C.

With the server running, open your web browser and type http://localhost:5000/ in the address bar. Figure 1 shows what you'll see after connecting to the application.



Hello World!

Figure 1: hello.py Flask application

If you type anything else after the base URL, the application will not know how to handle it and will return an error code 404 to the browser—the familiar error that you get when you navigate to a web page that does not exist.

3.5 Dynamic Routes

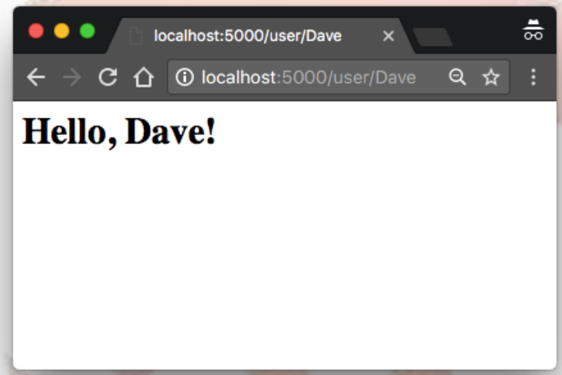
The second version of the application, shown in Example 2, adds a second route that is dynamic. When you visit the dynamic URL in your browser, you are presented with a personalized greeting that includes the name provided in the URL.

Example 2: hello.py: Flask application with a dynamic route

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

@app.route('/')

def index():

    return '<h1>Hello World!</h1>'
```

```
@app.route('/user/<name>')

def user(name):

    return '<h1>Hello, {}!</h1>'.format(name)
```

To test the dynamic route, make sure the server is running and then navigate to http://localhost:5000/user/Dave. The application will respond with the personalized greeting using the name dynamic argument. Try using different names in the URL to see how the view function always generates the response based on the name given. An example is shown in Figure 2:



Hello, Dave!

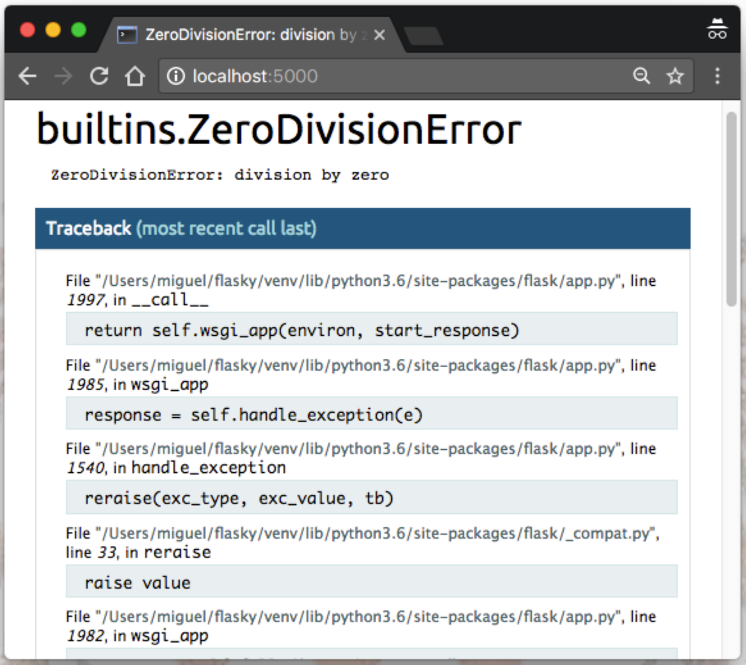
Figure 2: Dynamic route

3.6 Debug Mode

Flask applications can optionally be executed in debug mode. In this mode, two very convenient modules of the development server called the reloader and the debugger are enabled by default.

When the reloader is enabled, Flask watches all the source code files of your project and automatically restarts the server when any of the files are modified. Having a server running with the reloader enabled is extremely useful during development, because every time you modify and save a source file, the server automatically restarts and picks up the change.

The debugger is a web-based tool that appears in your browser when your application raises an unhandled exception. The web browser window transforms into an interactive stack trace that allows you to inspect source code and evaluate expressions in any place in the call stack. You can see how the debugger looks in Figure 3:



builtins.ZeroDivisionError

```
ZeroDivisionError: division by zero
```

Traceback (most recent call last)

File "/Users/miguel/flasky/venv/lib/python3.6/site-packages/flask/app.py", line 1997, in `__call__`

```
return self.wsgi_app(environ, start_response)
```

File "/Users/miguel/flasky/venv/lib/python3.6/site-packages/flask/app.py", line 1985, in `wsgi_app`

```
response = self.handle_exception(e)
```

File "/Users/miguel/flasky/venv/lib/python3.6/site-packages/flask/app.py", line 1540, in `handle_exception`

```
reraise(exc_type, exc_value, tb)
```

File "/Users/miguel/flasky/venv/lib/python3.6/site-packages/flask/_compat.py", line 33, in `reraise`

```
raise value
```

File "/Users/miguel/flasky/venv/lib/python3.6/site-packages/flask/app.py", line 1982, in `wsgi_app`

Figure 3: Flask Debugger

By default, debug mode is disabled. To enable it, set a FLASK_DEBUG=1 environment variable before invoking flask run:

```
(venv) $ export FLASK_APP=hello.py (venv) $ export FLASK_DEBUG=1
(venv) $ flask run

* Serving Flask app "hello"

* Forcing debug mode on

* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

* Restarting with stat

* Debugger is active!
* Debugger PIN: 273-181-528
```

If you are using Microsoft Windows, use set instead of export to set the environment variables.

3.7 The Request-Response Cycle

Now that you have played with a basic Flask application, you might want to know more about how Flask works its magic. The following sections describe some of the design aspects of the framework.

3.7.1 Application and Request Contexts

When Flask receives a request from a client, it needs to make a few objects available to the view function that will handle it. A good example is the request object, which encapsulates the HTTP request sent by the client.

The obvious way in which Flask could give a view function access to the request object is by sending it as an argument, but that would require every single view function in the application to have an extra argument. Things get more complicated if you consider that the request object is not the only object that view functions might need to access to fulfill a request.

To avoid cluttering view functions with lots of arguments that may not always be needed, Flask uses contexts to temporarily make certain objects globally accessible. Thanks to contexts, view functions like the following one can be written:

```
from flask import request @app.route('/')

def index():

    user_agent = request.headers.get('User-Agent')

    return '<p>Your browser is {}</p>'.format(user_agent)
```

Note how in this view function, request is used as if it were a global variable. In reality, request cannot be a global variable; in a multithreaded server several threads can be working on different requests from different clients all at the same time, so each thread needs to see a different object in request. Contexts enable Flask to make certain variables globally accessible to a thread without interfering with the other threads.

There are two contexts in Flask: the application context and the request context. Table 1 shows the variables exposed by each of these contexts.

Table 1: Flask Context Globals

Variable Name	Context	Description
current_app	Application Context	The application instance for the active application
g	Application Context	An object that the application can use for temporary storage during the handling of a request. This variable is reset with each request
request	Request Context	The request object, which encapsulates the content of an HTTP request sent by the client
session	Request Context	Then user session, a dictionary that the application can use to store values that are “remembered” between requests

Flask activates (or pushes) the application and request contexts before dispatching a request to the application and removes them after the request is handled. When the application context is pushed, the current_app and g variables become available to the thread. Likewise, when the request context is pushed, request and session become available as well. If any of these variables are accessed without an active application or request context, an error is generated. The four context variables will be covered in detail in this and later chapters, so don’t worry if you don’t understand why they are useful yet.

The following Python shell session demonstrates how the application context works:

```
>>> from hello import app

>>> from flask import current_app

>>> current_app.name

Traceback (most recent call last):

...

RuntimeError: working outside of application context
```

```
>>> app_ctx = app.app_context()

>>> app_ctx.push()

>>> current_app.name

'hello'

>>> app_ctx.pop()
```

In this example, current_app.name fails when there is no application context active but becomes valid once an application context for the application is pushed. Note how an application context is obtained by invoking app.app_context() on the application instance.

3.7.2 Request Dispatching

When the application receives a request from a client, it needs to find out what view function to invoke to service it. For this task, Flask looks up the URL given in the request in the application's URL map, which contains a mapping of URLs to the view functions that handle them. Flask builds this map using the data provided in the app.route decorator, or the equivalent non-decorator version, app.add_url_rule().

To see what the URL map in a Flask application looks like, you can inspect the map created for hello.py in the Python shell. Before you try this, make sure that your virtual environment is activated:

```
(venv) $ python

>>> from hello import app

>>> app.url_map

Map([<Rule '/' (HEAD, OPTIONS, GET) -> index>,

<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,

<Rule '/user/<name>' (HEAD, OPTIONS, GET) -> user>])
```

The / and /user/<name> routes were defined by the app.route decorators in the application. The /static/<filename> route is a special route added by Flask to give access to static files.

The (HEAD, OPTIONS, GET) elements shown in the URL map are the request methods that are handled by the routes. The HTTP specification defines that all requests are issued with a method, which normally indicates what action the client is asking the server to perform. Flask attaches methods to each route so that different request methods sent to the same URL can be handled by different view functions. The HEAD and OPTIONS methods are managed automatically by Flask, so in practice it can be said that in this application the three routes in the URL map are attached to the GET method, which is used when the client wants to request information such as a web page.

3.7.3 The Request Object

You have seen that Flask exposes a request object as a context variable named request. This is an extremely useful object that contains all the information that the client included in the HTTP request. Table 2 enumerates the most commonly used attributes and methods of the Flask request object.

Table 2: Flask Request Object

Attribute or Method	Description
form	A dictionary with all the form fields submitted with the request.
args	A dictionary with all the arguments passed in the query string of the URL.
values	A dictionary that combines the values in form and args.
cookies	A dictionary with all the cookies included in the request
headers	A dictionary with all the HTTP headers included in the request
files	A dictionary with all the file uploads included with the request.
get_data()	Returns the buffered data from the request body.
get_json()	Returns a Python dictionary with the parsed JSON included in the body of the request.
blueprint	The name of the Flask blueprint that is handling the request
endpoint	The name of the Flask endpoint that is handling the request. Flask uses the name of the view function as the endpoint name for a route.
method	The HTTP request method, such as GET or POST.
scheme	The URL scheme (http or https).
is_secure()	Returns True if the request came through a secure (HTTPS) connection.
host	The host defined in the request, including the port number if given by the client.
path	The path portion of the URL.
query_string	The query string portion of the URL, as a raw binary value.
full_path	The path and query string portions of the URL.
url	The complete URL requested by the client.

base_url	Same as url, but without the query string component.
remote_addr	The IP address of the client.
environ	The raw WSGI environment dictionary for the request.

3.7.4 Request Hooks

Sometimes it is useful to execute code before or after each request is processed. For example, at the start of each request it may be necessary to create a database connection or authenticate the user making the request. Instead of duplicating the code that performs these actions in every view function, Flask gives you the option to register common functions to be invoked before or after a request is dispatched.

Request hooks are implemented as decorators. These are the four hooks supported by Flask:

before_request

Registers a function to run before each request.

before_first_request

Registers a function to run only before the first request is handled. This can be a convenient way to add server initialization tasks.

after_request

Registers a function to run after each request, but only if no unhandled excep- tions occurred.

teardown_request

Registers a function to run after each request, even if unhandled exceptions occurred.

A common pattern to share data between request hook functions and view functions is to use the g context global as storage. For example, a before_request handler can load the logged-in user from the database and store it in g.user. Later, when the view function is invoked, it can retrieve the user from there.

Examples of request hooks will be shown in future chapters, so don't worry if the purpose of these hooks does not quite make sense yet.

3.7.5 Responses

When Flask invokes a view function, it expects its return value to be the response to the request. In most cases the response is a simple string that is sent back to the client as an HTML page.

But the HTTP protocol requires more than a string as a response to a request. A very important part of the HTTP response is the status code, which Flask by default sets to 200, the code that indicates that the request was carried out successfully.

When a view function needs to respond with a different status code, it can add the numeric code as a second return value after the response text. For example, the following view function returns a 400 status code, the code for a bad request error:

```
@app.route('/')

def index():

    return '<h1>Bad Request</h1>', 400
```

Responses returned by view functions can also take a third argument, a dictionary of headers that are added to the HTTP response.

Instead of returning one, two, or three values as a tuple, Flask view functions have the option of returning a response object. The make_response() function takes one, two, or three arguments, the same values that can be returned from a view function, and returns an equivalent response object. Sometimes it is useful to generate the response object inside the view function, and then use its methods to further configure the response. The following example creates a response object and then sets a cookie in it:

from flask import make_response @app.route('/')

```
def index():

    response = make_response('<h1>This document carries a
cookie!</h1>')

    response.set_cookie('answer', '42')

    return response
```

Table 3 shows the most commonly used attributes and methods available in response objects.

Table 3: Flask Response Object

Attribute/Method	Description
status_code	The numeric HTTP status code
headers	A dictionary-like object with all the headers that will be sent with the response
set_cookie()	Adds a cookie to the response
delete_cookie()	Removes a cookie
content_length	The length of the response body
content_type()	The media type of the response body
set_data()	Sets the response body as a string or bytes value
get_data()	Gets the response body

There is a special type of response called a redirect. This response does not include a page document; it just gives the browser a new URL to navigate to. A very common use of redirects is when working with web forms, as you will learn in Chapter 4.

A redirect is typically indicated with a 302 response status code and the URL to go to given in a Location header. A redirect response can be generated manually with a three-value return or with a response object, but given its frequent use, Flask provides a redirect() helper function that creates this type of response:

```
from flask import redirect @app.route('/')

def index():

return redirect('http://www.example.com')
```

Another special response is issued with the abort() function, which is used for error handling. The following example returns status code 404 if the id dynamic argument given in the URL does not represent a valid user:

```
from flask import abort @app.route('/user/<id>')

def get_user(id):

user = load_user(id)

if not user:

abort(404)

return '<h1>Hello, {}</h1>'.format(user.name)
```

Note that abort() does not return control back to the function because it raises an exception.

4. TEMPLATES

The key to writing applications that are easy to maintain is to write clean and well-structured code. The examples that you have seen so far are too simple to demonstrate this, but Flask view functions have two completely independent purposes disguised as one, which creates a problem.

The obvious task of a view function is to generate a response to a request, as you have seen in the examples shown in Section 2. For the simplest requests this is enough, but in many cases a request also triggers a change in the state of the application, and the view function is where this change is generated.

For example, consider a user who is registering a new account on a website. The user types an email address and a password in a web form and clicks the Submit button. On the server, a request with the data provided by the user arrives, and Flask dis- patches it to the view function that handles registration requests. This view function needs to talk to the database to get the new user added, and then generate a response to send back to the browser that includes a success or failure message. These two types of tasks are formally called business logic and presentation logic, respectively.

Mixing business and presentation logic leads to code that is hard to understand and maintain. Imagine having to build the HTML code for a large table by concatenating data obtained from the database with the necessary HTML string literals. Moving the presentation logic into templates helps improve the maintainability of the application.

A template is a file that contains the text of a response, with placeholder variables for the dynamic parts that will be known only in the context of a request. The process that replaces the variables with actual values and returns a final response string is called rendering. For the task of rendering templates, Flask uses a powerful template engine called Jinja2.

4.1. The Jinja2 Template Engine

In its simplest form, a Jinja2 template is a file that contains the text of a response. Example 3 shows a Jinja2 template that matches the response of the index() view function of Example 1.

Example 3. templates/index.html: Jinja2 template

```
<h1>Hello World!</h1>
```

The response returned by the user() view function of Example 2 has a dynamic component, which is represented by a variable. Example 4 shows the template that implements this response.

Example 4. templates/user.html: Jinja2 template

```
<h1>Hello, {{ name }}!</h1>
```

4.1.1 Rendering Templates

By default Flask looks for templates in a templates subdirectory located inside the main application directory. For the next version of hello.py, you need to create the templates subdirectory and store the templates defined in the previous examples in it as index.html and user.html, respectively.

The view functions in the application need to be modified to render these templates. Example 5 shows these changes.

Example 5. hello.py: rendering a template

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

@app.route('/')

def index():

    return render_template('index.html')

@app.route('/user/<name>')

def user(name):

return render_template('user.html', name=name)
```

The function render_template() provided by Flask integrates the Jinja2 template engine with the application. This function takes the filename of the template as its first argument. Any additional arguments are key-value pairs that represent actual values for variables referenced in the template. In this example, the second template is receiving a name variable.

Keyword arguments like name=name in the previous example are fairly common, but they may seem confusing and hard to understand if you are not used to them. The “name” on the left side represents the argument name, which is used in the place- holder written in the template. The “name” on the right side is a variable in the cur- rent scope that provides the value for the argument of the same name. While this is a common pattern, using the same variable name on both sides is not required.

4.1.2 Variables

The {{ name }} construct used in the template shown in Example 6 references a variable, a special placeholder that tells the template engine that the value that goes in that place should be obtained from data provided at the time the template is rendered.

Jinja2 recognizes variables of any type, even complex types such as lists, dictionaries, and objects. The following are some more examples of variables used in templates:

```
<p>A value from a dictionary: {{ mydict['key'] }}.</p>

<p>A value from a list: {{ mylist[3] }}.</p>

<p>A value from a list, with a variable index: {{

mylist[myintvar] }}.</p>

<p>A value from an object's method: {{ myobj.somemethod()

}}.</p>
```

Variables can be modified with filters, which are added after the variable name with a pipe character as separator. For example, the following template shows the name vari- able capitalized:

```
Hello, {{ name|capitalize }}
```

Table 4 below lists some of the commonly used filters that come with Jinja2.

Table 4: Jinja2 variable filters

Filter Name	Descriptions
safe	Renders the value without applying escaping
capitalize	Converts the first character of the value to uppercase and rest to lowercase
lower	Converts the value to lowercase characters
upper	Converts the value to uppercase characters
title	Capitalises each word in the value
trim	Removes leading and trailing whitespaces from the value
strlptags	Removes any HTML tags from the value before rendering

The safe filter is interesting to highlight. By default Jinja2 escapes all variables for security purposes. For example, if a variable is set to the value '<h1>Hello</h1>', Jinja2 will render the string as '&lt;h1&gt;Hello&lt;/h1&gt;', which will cause the h1 element to be displayed and not interpreted by the browser. Many times it is neces- sary to display HTML code stored in variables, and for those cases the safe filter is used.

The complete list of filters can be obtained from the official Jinja2 documentation.

4.1.3 Control Structures

Jinja2 offers several control structures that can be used to alter the flow of the tem- plate. This section introduces some of the most useful ones with simple examples.

The following example shows how conditional statements can be entered in a template:

```
{% if user %}

Hello, {{ user }}!

{% else %}

Hello, Stranger!

{% endif %}
```

Another common need in templates is to render a list of elements. This example shows how this can be done with a for loop:

```
<ul>

{% for comment in comments %}

<li>{{ comment }}</li>

{% endfor %}

</ul>
```

Jinja2 also supports macros, which are similar to functions in Python code. For example:

```
{% macro render_comment(comment) %}

<li>{{ comment }}</li>

{% endmacro %}

<ul>

{% for comment in comments %}

{{ render_comment(comment) }}

{% endfor %}

</ul>
```

To make macros more reusable, they can be stored in standalone files that are then imported from all the templates that need them:

```
{% import 'macros.html' as macros %}

<ul>

{% for comment in comments %}

{{ macros.render_comment(comment) }}
```

```
{% endfor %}

</ul>
```

Portions of template code that need to be repeated in several places can be stored in a separate file and included from all the templates to avoid repetition:

```
{% include 'common.html' %}
```

Yet another powerful way to reuse is through template inheritance, which is similar to class inheritance in Python code. First, a base template is created with the name base.html:

```
<html>

<head>

{% block head %}

<title>{% block title %}{% endblock %} - My Application</title>

{% endblock %}

</head>

<body>

{% block body %}

{% endblock %}

</body>

</html>
```

Base templates define blocks that can be overridden by derived templates. The Jinja2 block and endblock directives define blocks of content that are added to the base template. In this example, there are blocks called head, title, and body; note that title is contained by head. The following example is a derived template of the base template:

```
{% extends "base.html" %}

{% block title %}Index{% endblock %}

{% block head %}
```

```
{{ super() }}

<style>

</style>

{% endblock %}

{% block body %}

<h1>Hello, World!</h1>

{% endblock %}
```

The extends directive declares that this template derives from base.html. This directive is followed by new definitions for the three blocks defined in the base template, which are inserted in the proper places. When a block has some content in both the base and derived templates, the content from the derived template is used. Within this block, the derived template can call super() to reference the contents of the block in the base template. In the preceding example, this is done in the head block.

Real-world usage of all the control structures presented in this section will be shown later, so you will have the opportunity to see how they work.

4.2 Bootstrap Integration with Flask-Bootstrap

Bootstrap is an open-source web browser framework from Twitter that provides user interface components that help create clean and attractive web pages that are compatible with all modern web browsers used on desktop and mobile platforms.

Bootstrap is a client-side framework, so the server is not directly involved with it. All the server needs to do is provide HTML responses that reference Bootstrap's Cascading Style Sheets (CSS) and JavaScript files, and instantiate the desired user interface elements through HTML, CSS, and JavaScript code. The ideal place to do all this is in templates.

The naive approach to integrating Bootstrap with the application is to make all the necessary changes to the HTML templates, following the recommendations given by the Bootstrap documentation. But this is an area where the use of a Flask extension makes an integration task much simpler, while helping keep these changes nicely organized.

The extension is called Flask-Bootstrap, and it can be installed with pip:

```
(venv) $ pip install flask-bootstrap
```

Flask extensions are initialized at the same time the application instance is created. Example 6 shows the initialization of Flask-Bootstrap.

Example 6: hello.py: Flask-Bootstrap initialization

```
from flask_bootstrap import Bootstrap

# ...

bootstrap = Bootstrap(app)
```

The extension is usually imported from a flask_<name> package, where <name> is the extension name. Most Flask extensions follow one of two consistent patterns for initialisation. In Example 6, the extension is initialized by passing the application instance as an argument in the constructor.

Once Flask-Bootstrap is initialized, a base template that includes all the Bootstrap files and general structure is available to the application. The application then takes advantage of Jinja2's template inheritance to extend this base template. Example 7 shows a new version of user.html as a derived template.

Example 7: templates/user.html: template that uses Flask-Bootstrap

```
{% extends "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}

<div class="navbar navbar-inverse" role="navigation">

<div class="container">

<div class="navbar-header">

<button type="button" class="navbar-toggle"

data-toggle="collapse" data-target=".navbar-collapse">

<span class="sr-only">Toggle navigation</span>
```

```
<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

<a class="navbar-brand" href="/">Flasky</a>

</div>

<div class="navbar-collapse collapse">

<ul class="nav navbar-nav">

<li><a href="/">Home</a></li>

</ul>

</div>

</div>

</div>

{% endblock %}

{% block content %}

<div class="container">

<div class="page-header">

<h1>Hello, {{ name }}!</h1>

</div>

</div>

{% endblock %}
```

The Jinja2 extends directive implements the template inheritance by referencing bootstrap/base.html from Flask-Bootstrap. The base template from Flask-Bootstrap provides a skeleton web page that includes all the Bootstrap CSS and JavaScript files.

The user.html template defines three blocks called title, navbar, and content. These are all blocks that the base template exports for derived templates to define. The title block is straightforward; its contents will appear between <title> tags in the header of the rendered HTML document. The navbar and content blocks are reserved for the page navigation bar and main content.

In this template, the navbar block defines a simple navigation bar using Bootstrap components. The content block has a container <div> with a page header inside. The greeting line that was in the previous version of the template is now inside the page header. Figure 4 shows how the application looks with these changes.

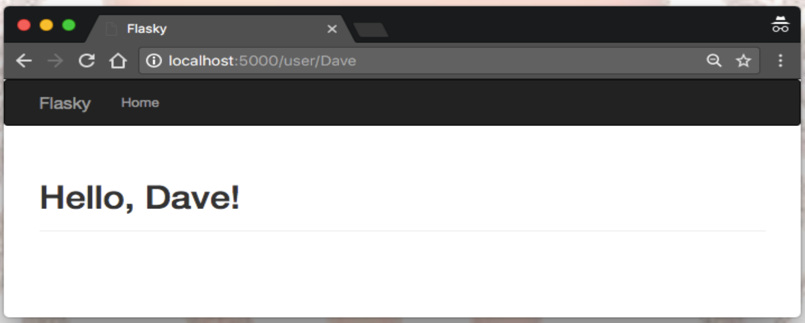


Figure 4. Bootstrap templates

Flask-Bootstrap's base.html template defines several other blocks that can be used in derived templates. Table 5 below shows the complete list of available blocks.

Table 5: Flask-Bootstrap's base template blocks

Block Name	Descriptions
doc	The entire HTML document
html_attribs	Attributes inside the <html> tag
html	The contents of the <html> tag
head	The contents of the <head> tag
title	The contents of the <title> tag
metas	The list of <meta> tags
styles	CSS definitions
body_attribs	Attributes inside the <body> tag
body	The contents of the <body> tag
navbar	User-defined navigation bar
content	User-defined page content
scripts	JavaScript declarations at the bottom of the document

Many of the blocks in the above Table are used by Flask-Bootstrap itself, so overriding them directly would cause problems. For example, the styles and scripts blocks are where the Bootstrap CSS and JavaScript files are declared. If the application needs to add its own content to a block that already has some content, then Jinja2's super() function must be used. For example, this is how the scripts block would need to be written in the derived template to add a new JavaScript file to the document:

```
{% block scripts %}

{{ super() }}

<script type="text/javascript" src="my-script.js"></script>

{% endblock %}
```

4.3 Custom Error Pages

When you enter an invalid route in your browser's address bar, you get a code 404 error page. Compared to the Bootstrap-powered pages, the default error page is now too plain and unattractive, and it has no consistency with the actual pages generated by the application.

Flask allows an application to define custom error pages that can be based on templates, like regular routes. The two most common error codes are 404, triggered when the client requests a page or route that is not known, and 500, triggered when there is an unhandled exception in the application. Example 8 shows how to provide custom handlers for these two errors using the app.errorhandler decorator.

Example 8: hello.py: custom error pages

```
@app.errorhandler(404)

def page_not_found(e):

return render_template('404.html'), 404

@app.errorhandler(500)

def internal_server_error(e):

return render_template('500.html'), 500
```

Error handlers return a response, like view functions, but they also need to return the numeric status code that corresponds to the error, which Flask conveniently accepts as a second return value.

The templates referenced in the error handlers need to be written. These templates should follow the same layout as the regular pages, so in this case they will have a navigation bar and a page header that shows the error message.

The straightforward way to write these templates is to copy templates/user.html to templates/404.html and templates/500.html and then change the page header elements in these two new files to the appropriate error messages, but this will generate a lot of duplication.

Jinja2's template inheritance can help with this. In the same way Flask-Bootstrap provides a base template with the basic layout of the page, the application can define its own base template with a uniform page layout that includes the navigation bar and leaves the page content to be defined in derived templates. Example 9 shows templates/base.html, a new template that inherits from bootstrap/base.html and defines the navigation bar but is itself a second-level base template to other templates such as templates/user.html, templates/404.html, and templates/500.html.

Example 9: templates/base.html: base application template with navigation bar

```
{% extends "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}

<div class="navbar navbar-inverse" role="navigation">

<div class="container">

<div class="navbar-header">

<button type="button" class="navbar-toggle"

data-toggle="collapse" data-target=".navbar-collapse">

<span class="sr-only">Toggle navigation</span>
```

```
<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

<a class="navbar-brand" href="/">Flasky</a>

</div>

<div class="navbar-collapse collapse">

<ul class="nav navbar-nav">

<li><a href="/">Home</a></li>

</ul>

</div>

</div>

</div>

{% endblock %}

{% block content %}

<div class="container">

{% block page_content %}{% endblock %}

</div>

{% endblock %}
```

The content block of this template is just a container <div> element that wraps a new empty block called page_content, which derived templates can define.

The templates of the application will now inherit from this template instead of directly from Flask-Bootstrap. Example 10 shows how simple it is to construct a custom code 404 error page that inherits from templates/base.html. The page for the 500 error is similar, and you can find it in the GitHub repository for the application.

Example 10: templates/404.html: custom code 404 error page using template inheritance

```
{% extends "base.html" %}

{% block title %}Flasky - Page Not Found{% endblock %}

{% block page_content %}

<div class="page-header">

<h1>Not Found</h1>

</div>

{% endblock %}
```

Figure 5 shows how the error page looks in the browser.

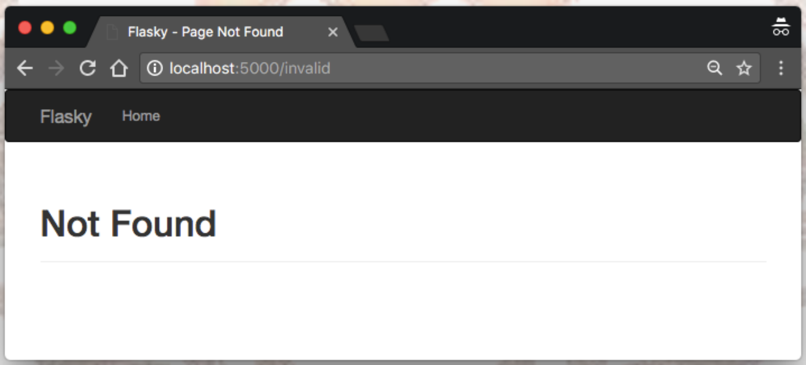


Figure 5: Custom code 404 error page

The templates/user.html template can now be simplified by making it inherit from the base template, as shown in Example 11.

Example 11:templates/user.html: simplified page template using template inheritance

```
{% extends "base.html" %}

{% block title %}Flasky{% endblock %}

{% block page_content %}

<div class="page-header">
```

```
<h1>Hello, {{ name }}!</h1>

</div>

{% endblock %}
```

4.4 Links

Any application that has more than one route will invariably need to include links that connect the different pages, such as in a navigation bar.

Writing the URLs as links directly in the template is trivial for simple routes, but for dynamic routes with variable portions it can get more complicated to build the URLs right in the template. Also, URLs written explicitly create an unwanted dependency on the routes defined in the code. If the routes are reorganized, links in templates may break.

To avoid these problems, Flask provides the url_for() helper function, which gener- ates URLs from the information stored in the application's URL map.

In its simplest usage, this function takes the view function name (or endpoint name for routes defined with app.add_url_route()) as its single argument and returns its URL. For example, in the current version of hello.py the call url_for('index') would return /, the root URL of the application. Calling url_for('index',

_external=True) would instead return an absolute URL, which in this example is

http://localhost:5000/.

Dynamic URLs can be generated with url_for() by passing the dynamic parts as keyword arguments. For example, url_for('user', name='john',

_external=True) would return http://localhost:5000/user/john.

Keyword arguments sent to url_for() are not limited to arguments used by dynamic routes. The function will add any arguments that are not dynamic to the query string. For example, url_for('user', name='john', page=2, version=1) would return /user/john?page=2&version=1.

4.5 Static Files

Web applications are not made of Python code and templates alone. Most applica- tions also use static files such as images, JavaScript source files, and CSS files that are all referenced from the HTML code in templates.

You may recall that when the hello.py application's URL map was inspected in Section 2, a static entry appeared in it. Flask automatically supports static files by adding a special route to the application defined as /static/<filename>. For example, a call to url_for('static', filename='css/styles.css', _external=True) would return http://localhost:5000/static/css/styles.css.

In its default configuration, Flask looks for static files in a subdirectory called static located in the application's root folder. Files can be organized in subdirectories inside this folder if desired. When the server receives a URL that maps to a static route, it generates a response that includes the contents of the corresponding file in the file system.

Example 12 shows how the application can include a favicon.ico icon in the base template for browsers to show in the address bar.

Example 12: templates/base.html: favicon definition

```
{% block head %}

{{ super() }}

<link rel="shortcut icon" href="{{ url_for('static',
filename='favicon.ico') }}" type="image/x-icon">

<link rel="icon" href="{{ url_for('static',
filename='favicon.ico') }}" type="image/x-icon">

{% endblock %}
```

The icon declaration is inserted at the end of the head block. Note how super() is used to preserve the original contents of the block defined in the base templates.

4.6 Localization of Dates and Times with Flask-Moment

Handling of dates and times in a web application is not a trivial problem when users work in different parts of the world.

The server needs uniform time units that are independent of the location of each user, so typically Coordinated Universal Time (UTC) is used. For users, however, seeing times expressed in UTC can be confusing, as users always expect to see dates and times presented in their local time and formatted according to the customs of their region.

An elegant solution that allows the server to work exclusively in UTC is to send these time units to the web browser, where they are converted to local time and rendered using JavaScript. Web browsers can do a much better job at this task because they have access to time zone and locale settings on the user's computer.

There is an excellent open source library written in JavaScript that renders dates and times in the browser called Moment.js. Flask-Moment is an extension for Flask applications that makes the integration of Moment.js into Jinja2 templates very easy. Flask- Moment is installed with pip:

```
(venv) $ pip install flask-moment
```

The extension is initialized in a similar way to Flask-Bootstrap. The required code is shown in Example 13.

Example 13: hello.py: initializing Flask-Moment

```
from flask_moment import Moment

moment = Moment(app)
```

Flask-Moment depends on jQuery.js in addition to Moment.js. These two libraries need to be included somewhere in the HTML document—either directly, in which case you can choose what versions to use, or through the helper functions provided by the extension, which reference tested versions of these libraries from a content delivery network (CDN). Because Bootstrap already includes jQuery.js, only Moment.js needs to be added in this case. Example 14 shows how this library is loaded in the scripts block of the template, while also preserving the original con- tents of the block provided by the base template. Note that since this is a predefined block in the Flask-Bootstrap base template, the location in templates/base.html where this block is inserted does not matter.

Example 14: templates/base.html: importing the Moment.js library

```
{% block scripts %}

{{ super() }}

{{ moment.include_moment() }}

{% endblock %}
```

To work with timestamps, Flask-Moment makes a moment object available to templates. Example 15 demonstrates passing a variable called current_time to the template for rendering.

Example 15: hello.py: adding a datetime variable

```
from datetime import datetime @app.route('/')

def index():

return    render_template('index.html',    current_time    =
datetime.utcnow())
```

Example 16 shows how this current_time template variable is rendered.

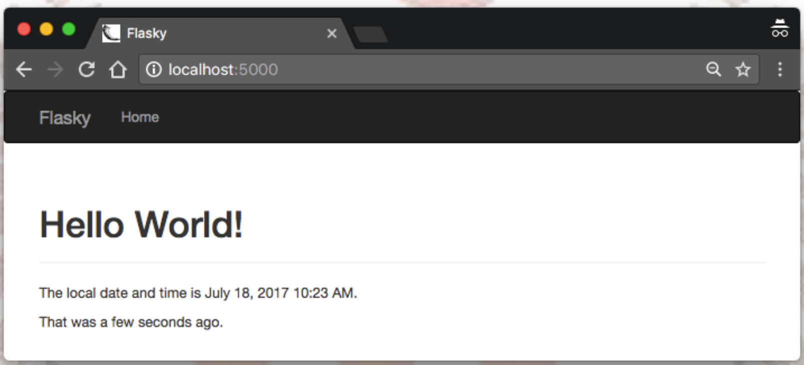
Example 16: templates/index.html: timestamp rendering with Flask-Moment

```
<p>The    local    date    and    time    is    {{
moment(current_time).format('LLL') }}.</p>

<p>That was {{ moment(current_time).fromNow(refresh=True) }}</p>
```

The format('LLL') function renders the date and time according to the time zone and locale settings in the client computer. The argument determines the rendering style, from 'L' to 'LLLL' for four different levels of verbosity. The format() function can also accept a long list of custom format specifiers.

The fromNow() render style shown in the second line renders a relative timestamp and automatically refreshes it as time passes. Initially this timestamp will be shown as "a few seconds ago," but the refresh=True option will keep it updated as time passes, so if you leave the page open for a few minutes you will see the text changing to "a minute ago," then "2 minutes ago," and so on.

Figure 6 shows how the http://localhost:5000/ route looks after the two timestamps are added to the index.html template.



Hello World!

The local date and time is July 18, 2017 10:23 AM.

That was a few seconds ago.

Figure 6: Page with two Flask-Moment timestamps

Flask-Moment implements the format(), fromNow(), fromTime(), calendar(), valueOf(), and unix() methods from Moment.js. Consult the Moment.js documen- tation to learn about all the formatting options offered by this library.

The timestamps rendered by Flask-Moment can be localized to many languages. A language can be selected in the template by passing the two-letter language code to function locale(), right after the Moment.js library is included. For example, here is how to configure Moment.js to use Spanish:

```
{% block scripts %}

{{ super() }}

{{ moment.include_moment() }}

{{ moment.locale('es') }}
```

```
{% endblock %}
```

With all the techniques discussed in this chapter, you should be able to build modern and user-friendly web pages for your application. The next chapter touches on an aspect of templates not yet discussed: how to interact with the user through web forms.

5. BUILDING A BASIC WEB APPLICATION: LOGIN AND REGISTRATION PAGE

5.1 Application Workflow

5.1.1 User Registration

In a web application, user registration is a crucial feature that allows new users to create an account. This typically involves collecting a username, password, and email address, and then storing this information securely in a database.

5.1.2 User Login

User login is the process by which users authenticate themselves to gain access to the web application. This involves verifying the username and password against the stored credentials in the database.

5.1.3 Session Management

Session management is essential for maintaining user state across multiple requests. When a user logs in, a session is created, and their information is stored, allowing the application to recognize the user on subsequent requests.

5.1.4 Templates

In Flask, templates are used to render HTML pages dynamically. This means that the HTML can include placeholders for dynamic content, which Flask fills in before sending the page to the user’s browser.

5.1.5 Static Files

Static files, such as CSS and JavaScript, are served by the Flask application to enhance the user interface and user experience.

5.2 Pre-requisites

- Knowledge of Python, MySQL Workbench, and basics of the Flask Framework.
- Python and MySQL Workbench installed on the system.
- A code editor like Visual Studio Code or Spyder.

5.3 Technologies Used

- Flask framework
- MySQL Workbench

5.4 Implementation of the Project

(1) Creating Environment

Step-1: Create an environment.

Create a project folder and a venv folder within.

```
py -3 -m venv venv
```

Step-2: Activate the environment.

```
venv\Scripts\activate
```

Step-3: Install Flask.

```
pip install Flask
```

(2) MySQL Workbench

Step-1: Install MySQL Workbench.

Step-2: Install mysqlbd module in your venv.

```
pip install flask-mysqldb
```

Step-3: Open MySQL Workbench.

Step-4: Write the following SQL code to create the database and table.

```
CREATE DATABASE geeklogin;
```

```
USE geeklogin;

CREATE TABLE accounts (

    id INT NOT NULL AUTO_INCREMENT,

    username VARCHAR(50) NOT NULL,

    password VARCHAR(50) NOT NULL,

    email VARCHAR(100),

    PRIMARY KEY (id)

);
```

Step-5: Execute the query.

(3) Creating Project

Step-1: Create an empty folder ‘login’.

Step-2: Open your code editor and open this ‘login’ folder.

Step-3: Create ‘app.py’ and write the following code:

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

from flask_mysqldb import MySQL

import MySQLdb.cursors

import re


app = Flask(__name__)

app.secret_key = 'your secret key'


app.config['MYSQL_HOST'] = 'localhost'

app.config['MYSQL_USER'] = 'root'

app.config['MYSQL_PASSWORD'] = 'your password'
app.config['MYSQL_DB'] = 'geeklogin'


mysql = MySQL(app)


@app.route('/')
@app.route('/login', methods=['GET', 'POST'])
def login():
    msg = ''
    if request.method == 'POST' and 'username' in request.form
and 'password' in request.form:
        username = request.form['username']
        password = request.form['password']
        cursor                                                  =
mysql.connection.cursor(MySQLdb.cursors.DictCursor)
        cursor.execute('SELECT * FROM accounts WHERE username
= %s AND password = %s', (username, password,))
        account = cursor.fetchone()
        if account:
            session['loggedin'] = True
            session['id'] = account['id']
            session['username'] = account['username']
            msg = 'Logged in successfully!'
            return render_template('index.html', msg=msg)
        else:
            msg = 'Incorrect username/password!'
    return render_template('login.html', msg=msg)
```

```
@app.route('/logout')
def logout():
    session.pop('loggedin', None)
    session.pop('id', None)
    session.pop('username', None)
    return redirect(url_for('login'))

@app.route('/register', methods=['GET', 'POST'])
def register():
    msg = ''
    if request.method == 'POST' and 'username' in request.form
and 'password' in request.form and 'email' in request.form:
        username = request.form['username']
        password = request.form['password']
        email = request.form['email']
        cursor                                                  =
mysql.connection.cursor(MySQLdb.cursors.DictCursor)
        cursor.execute('SELECT * FROM accounts WHERE username
= %s', (username,))
        account = cursor.fetchone()
        if account:
            msg = 'Account already exists!'
        elif not re.match(r'[^@]+@[^@]+\.[^@]+', email):
            msg = 'Invalid email address!'
        elif not re.match(r'[A-Za-z0-9]+', username):
            msg = 'Username must contain only characters and
numbers!'
        elif not username or not password or not email:
            msg = 'Please fill out the form!'
        else:
            cursor.execute('INSERT INTO accounts VALUES (NULL,
%s, %s, %s)', (username, password, email,))
            mysql.connection.commit()
            msg = 'You have successfully registered!'
    elif request.method == 'POST':
        msg = 'Please fill out the form!'
    return render_template('register.html', msg=msg)


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

Step-4: Create the folder ‘templates’ and add login.html, register.html, and index.html.

login.html code:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Login</title>
    <link rel="stylesheet" href="{{ url_for('static',
filename='style.css') }}">
</head>
<body>
```

```
    <div align="center">
        <div class="border">
            <div class="header">
                <h1 class="word">Login</h1>
            </div>
            <form action="{{ url_for('login') }}" method="post">
                <div class="msg">{{ msg }}</div>
                <input id="username" name="username" type="text"
placeholder="Enter Your Username" class="textbox"/><br><br>
                <input id="password" name="password"
type="password" placeholder="Enter Your Password"
class="textbox"/><br><br>
                <input type="submit" class="btn" value="Sign
In"><br><br>
            </form>
            <p class="bottom">Don't have an account? <a href="{{
url_for('register') }}">Sign Up here</a></p>
        </div>
    </div>
</body>
</html>
```

register.html code:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Register</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
    <div align="center">
        <div class="border">
            <div class="header">
                <h1 class="word">Register</h1>
            </div>
            <form action="{{ url_for('register') }}" method="post">
                <div class="msg">{{ msg }}</div>
                <input id="username" name="username" type="text" placeholder="Enter Your Username" class="textbox"/><br><br>
                <input id="password" name="password" type="password" placeholder="Enter Your Password" class="textbox"/><br><br>
                <input id="email" name="email" type="text" placeholder="Enter Your Email ID" class="textbox"/><br><br>
                <input type="submit" class="btn" value="Sign Up"><br>
            </form>
            <p class="bottom">Already have an account? <a href="{{ url_for('login') }}">Sign In here</a></p>
        </div>
    </div>
```

```
</body>

</html>
```

index.html code:

```
<!DOCTYPE html>

<html>

<head>

    <meta charset="UTF-8">

    <title>Index</title>

    <link     rel="stylesheet"     href="{{     url_for('static',
filename='style.css') }}">

</head>

<body>

    <div align="center">

        <div class="border">

            <div class="header">

                <h1 class="word">Index</h1>

            </div>

            <h1  class="bottom">Hi  {{session.username}}!!  Welcome
to the index page...</h1><br><br>

            <a      href="{{       url_for('logout')       }}"
class="btn">Logout</a>

        </div>

    </div>

</body>

</html>
```

Step-5: Create the folder ‘static’ and add style.css file.

style.css code:

```
.header{

    padding: 5px 120px;

    width: 150px;

    height: 70px;

    background-color: #236B8E;

}

.border{

    padding: 80px 50px;

    width: 400px;

    height: 450px;

    border: 1px solid #236B8E;

    border-radius: 0px;

    background-color: #9AC0CD;

}

.btn {

    padding: 10px 40px;

    background-color: #236B8E;

    color: #FFFFFF;

    font-style: oblique;

    font-weight: bold;

    border-radius: 10px;

}

.textbox{

    padding: 10px 40px;
    background-color: #236B8E;
    text-color: #FFFFFF;
    border-radius: 10px;
}
::placeholder {
    color: #FFFFFF;
    opacity: 1;
    font-style: oblique;
    font-weight: bold;
}
.word{
    color: #FFFFFF;
    font-style: oblique;
    font-weight: bold;
}
.bottom{
    color: #236B8E;
    font-style: oblique;
    font-weight: bold;
}
```

Step-6: The project structure should look like this:

```
login/
├── app.py
├── templates/
|   ├── login.html
```

```
│   ├── register.html
│   ├── index.html
└── static/
    └── style.css
```

(4) Run the Project

Step-1: Run the server.

```
python app.py
```

Step-2: Browse the URL localhost:5000.

(5) Testing of the Application

Step-1: If you are a new user, go to the sign-up page and fill in the details.

Step-2: After registration, go to the login page. Enter your username and password and sign in.

Step-3: If your login is successful, you will be moved to the index page and your name will be displayed.

6. SUMMARY

After an immersive journey through Python for Web Development using Flask, you've now reached the end of Unit 11. Here's a wrap-up of what we've covered and how these elements come together in real-world applications.

Starting with the basics, we dove into setting up your development environment. By now, you understand the importance of virtual environments and using pip for managing Python packages—skills that are crucial not just for Flask but for Python programming in general. The setup process might have seemed meticulous, but it ensures that your projects are manageable and isolated from global Python settings, which is vital for maintaining project-specific dependencies.

As we moved into the core of Flask, you learned about the architecture of a Flask application. Initialization, routing, and view functions are the backbone of how Flask serves web pages. You've seen how simple it is to start a development server and how routes are created to respond to user requests. Each piece of code you wrote linked together to form a more comprehensive understanding of Flask's capabilities and how dynamic routes and debug mode enhance the development process. By experimenting with these features, you gained a hands-on appreciation of Flask's flexibility and power.

The introduction of templates through Jinja2 might have been one of the more enlightening sections. Templates are powerful tools that help separate business logic from presentation in your applications, adhering to the DRY (Don't Repeat Yourself) principle. You practiced rendering templates and passing data to them, which is pivotal for creating dynamic websites. Moreover, incorporating control structures into templates allowed you to dynamically display content based on conditions, which is a common requirement in real applications.

On the more advanced side, you tackled user authentication, role management, and even blogging features within our social blogging application. These modules not only added complexity but also practicality to your learning experience, showing how Flask can be used to build robust applications with user interactions at their core. The concepts of database interactions, user comments, and following mechanisms highlighted how data can be manipulated and displayed in a community-driven project.

Finally, the dive into RESTful web services with Flask illustrated how you could extend the reach of your applications to interact seamlessly with other services and provide APIs that others might consume.

Throughout this unit, the goal was not just to teach you how to use Flask, but to inspire you to think about web applications as a cohesive blend of technology, user experience, and security. With these skills, you're now better equipped to tackle web development projects that are secure, efficient, and user-friendly. Remember, the world of web development is ever-evolving, and continuous learning will be key to your success and adaptability in the field.

7. SELF-ASSESSMENT QUESTIONS

1. What is the purpose of a virtual environment in Flask development?
   - A. To speed up the website
   - B. To isolate project dependencies
   - C. To increase the security of the web server
   - D. None of the above
2. Which command is used to install Flask in a virtual environment?
   - A. pip install django
   - B. install flask
   - C. pip install flask
   - D. setup flask
3. What does the Flask route() decorator do?
   - A. Styles the webpage
   - B. Connects a URL rule to a Python function
   - C. Sends email notifications
   - D. None of the above
4. In Flask, what is the default function to start the local development server?
   - A. run()
   - B. start()
   - C. serve()
   - D. execute()
5. What type of files does the static directory in a Flask application typically contain?
   - A. Python scripts
   - B. Configuration files
   - C. CSS, JavaScript, and image files
   - D. Database files
6. Which of the following is a Jinja2 template feature used for reusing parts of a template?
   - A. Functions
   - B. Macros
   - C. Classes
   - D. Modules

7. What is used in Flask to check user authentication status?
   A. @authenticate
   B. @login_required
   C. @check_user
   D. @user_status
8. Which HTTP method is primarily used to submit web forms?
   A. GET
   B. POST
   C. PUT
   D. DELETE
9. Flask extensions like Flask-Login are used for:
   A. Database migrations
   B. Enhancing application features like handling user sessions
   C. Improving template rendering speed
   D. Configuring the server
10. In which Flask function do you typically commit database transactions?
    A. commit()
    B. save()
    C. push()
    D. update()
11. To create a virtual environment in Python, use the command ______.
12. Flask applications are instances of the class Flask(__name__) where __name__ is usually the name of the application's ______ module.
13. To add a URL rule in Flask, you define a ______ followed by the function that returns the webpage content.
14. In Flask, ______ files are used to add dynamic content to HTML pages.
15. To connect to a database in Flask, you often use the ______ object which represents the database session.
16. The configuration settings for a Flask application are usually stored in the ______ object.
17. ______ are special templates in Flask used when a certain type of HTTP error occurs.

18. To protect a route so that only logged-in users can access it, the decorator ______ is commonly used.
19. A Flask extension that can handle password hashing securely is Flask-______.
20. The Flask ______ function is used to terminate a session and clean up before the application closes.

8. TERMINAL QUESTIONS

1. Explain the benefits of using a virtual environment for Flask development.
2. Describe the role of the Flask route() decorator and how it interacts with view functions.
3. What are the different parts of a Flask application and their purposes?
4. Discuss the significance of the request-response cycle in Flask.
5. Explain the concept of context in Flask and differentiate between application and request contexts.
6. Describe how Flask uses routing to decide which piece of code should handle an incoming request.
7. How does Flask handle client-server data transmission?
8. Discuss the importance of data consistency in Flask and how transactions are used to achieve it.
9. What are templates in Flask, and how do they integrate with the Jinja2 template engine?
10. Explain the process and importance of user authentication in a Flask web application.
11. Write a Flask route to handle the homepage of a website that returns "Welcome to My Site!".
12. Create a Flask application instance and define a route that returns the string "Hello, World!" when visiting the root URL.
13. Demonstrate how to use Flask's url_for() to generate URLs for a specific function named profile that accepts a username as an argument.
14. Write a Flask route and a corresponding view function that accepts a user ID as a parameter and returns "User ID is: [id]".
15. Implement a simple login form handling route using Flask that checks if username and password are "admin".
16. Write a Flask application that connects to an SQLite database named app.db and retrieves all records from the table users.
17. Using Flask, create a route that allows users to submit a comment. The route should handle both GET and POST requests.
18. Implement a Flask route that uses a Jinja2 template to display a list of items passed as a context variable to the template.

19. Write a Python function using Flask that performs a database transaction involving updating a user's email and committing the transaction.
20. Develop a Flask route that demonstrates the use of session management to store a user's login status after authentication.

9. ANSWERS

9.1 Self-assessment Questions

1. B) To isolate project dependencies
2. C) pip install flask
3. B) Connects a URL rule to a Python function
4. A) run()
5. C) CSS, JavaScript, and image files
6. B) Macros
7. B) @login_required
8. B) POST
9. B) Enhancing application features like handling user sessions
10. A) commit()
11. python -m venv env
12. main
13. route
14. template
15. session
16. app.config
17. custom error pages
18. @login_required
19. Flask-Bcrypt
20. close()

9.2 Terminal Questions

1. Virtual Environments - The benefits of using a virtual environment for Flask development can be found in Section 1.3 of the unit.
2. Routes and View Functions - Details on Flask's route() decorator and view functions can be referenced in Section 2.2.
3. Parts of a Flask Application - The different parts and their purposes are explained in Section 2.1.

4. The Request-Response Cycle - This process is thoroughly described in Section 2.7.
5. Application and Request Contexts - The concept of context in Flask is covered in Section 2.9.
6. Request Dispatching - How Flask uses routing is detailed in Section 2.8.
7. The Request Object - Handling of client-server data transmission via the request object is in Section 2.9.
8. Transactions - The importance of data consistency and transaction use in Flask are part of Section 2.4.
9. Templates - Explanation of templates and integration with Jinja2 is found in Section 3.
10. User Authentication - Details on user authentication processes are located in Section 4.1 and Section 4.6.
11. 11-12. Basic Routing - Creating basic routes and a Flask application instance are explained in Section 2.2.
12. 13-14. Dynamic Routes and URL Generation - Using url_for() and parameter handling in routes can be referenced in Section 2.6.
13. 15-16. Form Handling and Database Connection - Form handling and connecting to SQLite are covered in Section 2.3 and Section 6.
14. 17-18. GET and POST Handling, Jinja Templates - Handling different request methods and using Jinja2 templates are discussed in Section 2.3 and Section 3.2.
15. 19-20. Database Transactions and Session Management - Implementing transactions and managing sessions using Flask are elaborated in Section 2.4 and Section 4.6.

10. REFERENCES

- Grinberg, M. (2018). *FLASK WEB DEVELOPMENT : developing web applications with python.*