Flask

* Flask is a web framework that provides libraries to build lightweight web applications in python.
* It is developed by **Armin Ronacher** in 2004,Released in April 2010.
* Flask is a part of pallets project(commonly known as POCOO).
* Flask can be used to serve Single-Page Applications (SPA).
* Flask is considered as a microweb framework.

It is classified as a microweb framework because it does not require tools or libraries. It has no database abstraction layer, form validation,authentication or any other components.

It is based on WSGI toolkit and jinja2 template engine.

What is Jinja2?

Jinja2 is a web template engine which combines a template with a certain data source to render the dynamic web pages.(in html page you can write a python code using jinja)

What is Werkzeug?

Werkzeug is a utility library for the python programming language for web server gateway interface(WSGI) application . It is considered as the specification for the universal interface/interaction between the web server and web application.

MarkupSafe🡪string handling library

(MarkupSafe implements a text object that escapes characters so it is safe to use in HTML and XML. Characters that have special meanings are replaced so that they display as the actual characters.)

ItsDangerous🡪safe data serialization library.

(Sometimes you want to send some data to untrusted environments, then get it back later. To do this safely, the data must be signed to detect changes.)

Features

1. Flask provides a **development server** and a **debugger**.
2. It uses **Jinja2** templates.
3. **Werkzeug** for **WSGI 1.0** support.
4. It provides integrated support for **unit testing**.
5. Many extensions are available for Flask, which can be used to enhance its functionalities.
6. Inbuilt server 5000.
7. It support secure Cookies at client side.

Applications of Flask:

LinkedIn, Pinterest, etc.

What is Virtual Environments?

1. A virtual environment is a tool that helps to keep dependencies required by different projects separate by creating isolated python virtual environments for them. This is one of the most important tools that most Python developers use..

**Steps:-**

1.pip install flask in terminal.

2. Create a python package by name com inside the folder.

3.Inside com, create a .py file in which we will write a flask code.

1st Application :

1. **from** flask **import** Flask
3. app = Flask(\_\_name\_\_) #creating the Flask class object
4. #\_\_name\_\_ 🡪 it holds the name of current modules
5. @app.route('/') # / 🡪indicates home page
6. **def** home():
7. **return** "hello, this is our first flask website";
9. **if** \_\_name\_\_ =='\_\_main\_\_':
10. app.run()  # To run a flask application

To build the python web application, we need to import the Flask module. An object of the Flask class is considered as the WSGI application.

We need to pass the name of the current module, i.e. \_\_name\_\_ as the argument into the Flask constructor.

The route() decorator in Flask is used to bind an URL to a function. As a result when the URL is mentioned in the browser, the function is executed to give the result.

The syntax is given below.

app.route(rule, options)

It accepts the following parameters.

1. rule: It represents the URL binding with the function.
2. options: It represents the list of parameters to be associated with the rule object

As we can see here, the / URL is bound to the main function which is responsible for returning the server response. It can return a string to be printed on the browser's window or we can use the HTML template to return the HTML file as a response from the server.

Finally, the run method of the Flask class is used to run the flask application on the local development server.

The syntax is given below.

app.run(host, port, debug, options)

|  |  |  |
| --- | --- | --- |
| **SN** | **Option** | **Description** |
| 1 | Host | The default hostname is 127.0.0.1, i.e. localhost. |
| 2 | Port | The port number to which the server is listening to. The default port number is 5000. |
| 3 | Debug | The default is false. It provides debug information if it is set to true. |
| 4 | Options | It contains the information to be forwarded to the server. |

Debug mode

A **Flask** application is started by calling the **run()** method. However, while the application is under development, it should be restarted manually for each change in the code. To avoid this inconvenience, enable **debug support**. The server will then reload itself if the code changes. It will also provide a useful debugger to track the errors if any, in the application.

The **Debug** mode is enabled by setting the **debug** property of the **application** object to **True** before running or passing the debug parameter to the **run()** method.

app.run(debug=True)

or

app.debug=True

app.run()

**Template :-** A template in Flask is basically written in HTML, CSS, and Javascript in a .html file. Flask framework efficiently handles and generates dynamically HTML web pages that are visible to the end-user.

**Steps:-**

1.Create a new Directory by name templates inside com package.

2. Create .html file in this templates directory.

3.If we change the name of directory then set template\_folder = ‘dir\_name’ in flask class constructor.

4.Render the html file by **render\_template** function.

1. **from** flask **import** Flask, render\_template
3. app = Flask(\_\_name\_\_)
5. @app.route('/v1')
6. **def** view1():
7. **return** render\_template(".html filename")
9. **if** \_\_name\_\_ =='\_\_main\_\_':
10. app.run(debug=True)

**Context:-**

**Steps:-**

1.context is send as a keyword arguments in flask.

2. In frontend, we have to access by the key name using jinja tags

{{}}-🡪to Display

{% %}-🡪for code

1. **from** flask **import** Flask, render\_template
3. app = Flask(\_\_name\_\_)
5. @app.route('/v1')
6. **def** view1():
7. **name=input(“enter name:”)**
8. **Emp = [‘anu’,’abhi’,’apu’,’om’,’ sid’]**
9. **return** render\_template(".html filename",nm=name,emp=Emp)
11. **if** \_\_name\_\_ =='\_\_main\_\_':
12. app.run(debug=True)

.html file

<h1>Name of user:{{nm}}</h1>

<h1>list of employee:{{li}}</h1>

<h1>List </h1>

{% for i in emp %}

<h2>{{i}}</h2>

{% endfor %}

**Static Files: -**

Static files are files that don't change when your application is running. Static Files such as Images, CSS or JS files are often loaded via a different app in production websites to avoid loading multiple stuff from the same server.

Steps:-

1.create a directory by name static inside com pkg.

2. Inside static dir create dir by name image, css, js.

3.If we change the name of folder the you set static\_folder="dir\_nm" in flask class constructor.

1. **from** flask **import** Flask, render\_template
2. app = Flask(\_\_name\_\_)
3. @app.route('/v1')
4. **def** view1():
5. **return** render\_template("filename.html ")
7. **if** \_\_name\_\_ =='\_\_main\_\_':
8. app.run(debug=True)

.html file

<html>

<head>

<title>Static file</title>

<link rel = ”stylesheet” href=”/static/css/.css filename”>#currently present in template directory

</head>

<body>

<h1>Hello</h1>

<h2>Hello1</h2>

<h3>Hello2</h3>

<h4>Hello3</h4>

<img src=”/static/img/.img file”>

<script type=”text/javascript” src=” /static/js/file\_nm.js”></script>

</body>

</html>

.css File

h1{  
 color: red;

}

h2{  
 color: blue;

}

.js file

Alert(“my document”)

**GET REQUEST**

Steps:

1.import request

2.To fetch the frontend data into backend using **request.args** (dictonary).

1. **from** flask **import** Flask, render\_template, request
3. app = Flask(\_\_name\_\_)
5. @app.route('/v1')
6. **def** view1():
8. **return** render\_template(".html filename”)
10. @app.route('/v2')
11. **def** view2():
12. unm = request.args.get(“u”)
13. pwd = request.args.get(‘p’)
14. Print(request.args)  #return immutablemultidict
15. Print(f”username:{unm}\npassword:{pwd}”)
16. **return** (“Successfully logged in!!”)
17. **if** \_\_name\_\_ =='\_\_main\_\_':
18. app.run(debug=True)

.html file

<html>

<head>

<title>LOGIN GET</title>

</head>

<body>

<h1>Get Request Form</h1>

<form action=”/v2”>

Name:-<input type=”text” name=”n”>

Password:-<input type="password" name="p”>

<input type="submit”>

</form>

</body>

</html>

**POST REQUEST**

Steps:

1.import request

2.to add method type explicitly in route()method

3.To fetch the frontend data into backend using **request.form** (dictonary).

1. **from** flask **import** Flask, render\_template, request
3. app = Flask(\_\_name\_\_)
5. @app.route('/v1')
6. **def** view1():
8. **return** render\_template("first.html filename”)
10. @app.route('/v2',methods=[“POST”])#explicitly implement type of method
11. **def** view2():
12. unm = request.form.get(“u”)
13. pwd = request.form.get(‘p’)
14. Print(request.form)
15. Print(f”username:{unm}\npassword:{pwd}”)
16. **return** render\_template(“sec.html”)
18. **if** \_\_name\_\_ =='\_\_main\_\_':
19. app.run(debug=True)

first.html file

<html>

<head>

<title>LOGIN GET</title>

</head>

<body>

<h1>Get Request Form</h1>

<form action=”/v2” method= “post”>

Name:-<input type=”text” name= “n”>

Password:-<input type="password" name="p”>

<input type="submit”>

</form>

</body>

</html>

sec.html file

<html>

<head>

<title>LOGIN GET</title>

</head>

<body>

<h1>Successfully Loged In!!!</h1>

<h2>Username: {{request.form.get(“n”)}} </h2>

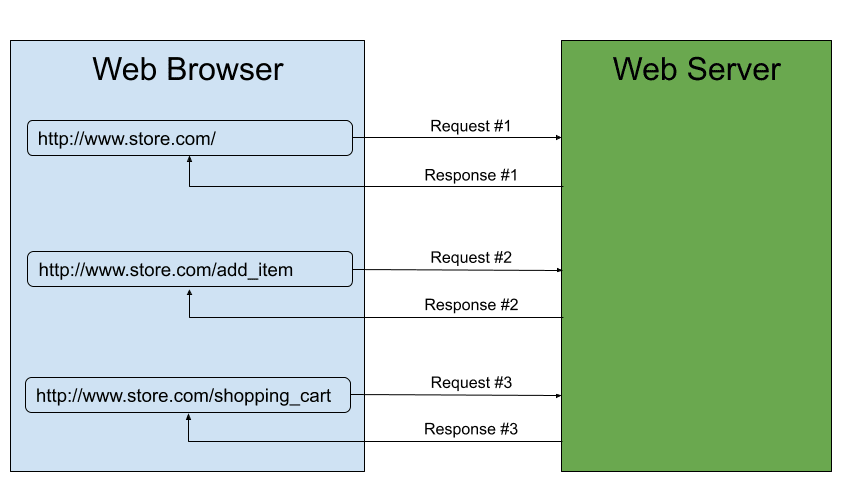
</form>

</body>

</html>

**Session Management**

Since HTTP is a stateless protocol, each request has no knowledge of any requests previously executed:



In any web application we send the request to server and we receive response from there using http protocol. Http protocol is stateless protocol and to make it stateful we use session management techniques to hold the information.

**There are 3 techniques:**

1.Hidden form field

2.Cookies

3.Session

**Hidden Form Field:**

* It is the simplest technique of all as it is a html tag.
* There is no overhead on server and client(browser) as data is hold inside an input field.

Disadvantages:

* Code redundancy will increase as we have to write the hidden tag in form always.
* Data is not secure.
* We cannot carry object.

**Syntax:**

<input type=’hidden’ name=’d1’ value=’{{request.args.get(‘d1’)}}’>

**Cookies:**

* A cookie is stored on a client’s computer in the form of a text file. Its purpose is to remember and track data pertaining to a client’s usage for better visitor experience and site statistics.
* A **Request object** contains a cookie’s attribute. It is a dictionary object of all the cookie variables and their corresponding values, a client has transmitted. In addition to it, a cookie also stores its expiry time, path and domain name of the site.
* In Flask, cookies are set on response object. Use **make\_response()** function it return a response Object using response object we call set\_cookies() . After that, use the **set\_cookie()** function of response object to store a cookie.

**There are 2 types of cookies:**

1.**In memory cookies** – Temporary cookies which are cleared when browser is closed.

2.**Persistant cookies** – Permanent cookies which are cleared when their time limit ends.

Syntax to set cookie:

@app.route("/first")

def m2():

res = make\_response(render\_template("second.html"))

res.set\_cookie("fd",request.args.get("d1"))

return res

Syntax to get cookie data:

First Data: {{request.cookies.get("fd")}}

@app.route("/first")  
def m2():  
 res = make\_response(render\_template("second.html"))  
 res.set\_cookie("fd",request.args.get("d1"))  
 return res

Syntax to get cookie data:

First Data: {{request.cookies.get("fd")}}

Syntax to set time limit:

res.set\_cookie ("fd”, request.args.get("d1"), max\_age = 60\*60\*24\*365\*5)

#sec\*min\*hrs\*days\*years

Syntax to delete a cookie:

res.set\_cookie ("fd", request.cookies.get("fd"), max\_age = 0)

**Advantages:**

* Code redundancy will reduce.
* There is no overhead on server.

**Disadvantages:**

* There is overhead on client but if browser cookie is disabled then this technique will not work.
* Data is not secure as it is stored on client machine.
* They cannot carry object; they can carry only string.

**Session**

* The concept of a session is very much similar to that of a cookie. However, the session data is stored on the server.
* The session can be defined as the duration for which a user logs into the server and logs out.
* The session data is stored in cryptographic format.
* A secret key is defined mandatory if we want to use session.
* In the flask, a session object is used to track the session data which is a dictionary object that contains a key-value pair of the session variables and their associated values.
* A session with each client is assigned a **Session ID**.

**Syntax:**

**session[key] = value**

for e.g,

session['fd'] = request.args.get("d1")

syntax to get cookies:

First Data: {{session.get("fd")}}

**Flash Message:**

1.Flash is a function which is used to carry bunch of message inside template.

2.In background, flash message are stored in session.

3.we need to define a secret key to use session.

4.we can send flash message with or without category.

First, make sure you have the flash module imported from the flask package:

1. **from** flask **import** Flask, render\_template, **flash**
3. app = Flask(\_\_name\_\_)
5. @app.route("/")
6. def withoutcategory():
7. **#flash(“messages”)**
8. flash("Flash message")
9. flash("Flash message1")
10. flash("Flash message2")
11. **return** render\_template("first.html filename”)
12. @app.route("/v1")
13. def withcategory():
14. **#flash(“messages”,”error”)**
15. flash("error","ERROR")
16. flash("Information","Information")
17. flash("Warning","warning")
18. **return** render\_template("first.html filename”)
20. **if** \_\_name\_\_ =='\_\_main\_\_':
21. app.run(debug=True)

withcategory.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Flash\_message</title>  
</head>  
<body>  
 {% for i,j in get\_flashed\_messages(with\_categories=True) %}  
 <h1>{{i}}----{{j}}</h1>  
 {% endfor %}  
  
</body>  
</html>

Withoutcategory.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Flash\_message</title>  
</head>  
<body>  
 {% for i in get\_flashed\_messages() %}  
 <h1>{{i}}</h1>  
 {% endfor %}  
  
</body>  
</html>

* In the above code, we're using the flash function to store success or error messages depending on the login result. The first argument to the flash function is the message string, and the second argument is the category (e.g., 'success', 'error').
* In your HTML template (e.g., login.html), you can retrieve and display flash messages using the get\_flashed\_messages function.
* The get\_flashed\_messages function retrieves the flashed messages from the session. In the template code above, we iterate over the messages and display them in a list. You can style the messages according to your preferences.

**Blueprint:**

1.It is a class in Flask which is used to divide our project into different modules.

2.It helps to easily maintain the project .

3.We can divide our project into different modules like login, logout, admin, client, etc.

A screenshot of a computer program

Description automatically generated

**Directory Structure**

Main.py file

1.Import all blueprint module and then register to your main.py file.

2.  Flask Blueprint, you have to import it and then **register** it in the application using **register\_blueprint().** When a Flask Blueprint is registered, the application is extended with its contents.

from flask import Flask  
from com.admin import Admin  
from com.client import Client  
  
  
app = Flask(\_\_name\_\_)

#register Blueprint module  
app.register\_blueprint(Admin.aobj)#in parameter you pass blueprint object  
app.register\_blueprint(Client.cobj)  
  
@app.route("/")  
def main():  
 return "This is my main File"

if \_\_name\_\_=="\_\_main\_\_":  
 app.run(debug=True)

Admin.py

1. **Create** a Blueprint object called**aobj**.
2. **Add** views to **aobj** using the route decorator.
3. **It contain 3 parameters:**
   1. **Blueprint name (adminblueprint)-** The name of the blueprint. Will be prepended to each endpoint name.
   2. **Import module name (\_\_name\_\_)-** The name of the blueprint package, usually \_\_name\_\_. This helps locate the root\_path for the blueprint.
   3. **Add any static,template,etc.-** A folder with templates that should be added to the app's template search path. The path is relative to the blueprint's root path. Blueprint templates are disabled by default. Blueprint templates have a lower precedence than those in the app's templates folder.

from flask import Blueprint,render\_template  
  
  
#create object of Blueprint class  
aobj = Blueprint("adminblueprint",\_\_name\_\_,template\_folder="templates")  
  
  
@aobj.route("/av")  
def view1():  
 return render\_template("a.html")

a.html

<!DOCTYPE html>  
<head>  
 <meta charset="UTF-8">  
 <title>ADMIN</title>  
</head>  
<body>  
 <h1>This is my Admin page</h1>  
  
</body>  
</html>

Client.py

from flask import Blueprint,render\_template  
  
cobj = Blueprint("clientblueprint",\_\_name\_\_,template\_folder="templates")  
  
  
@cobj.route("/cv")  
def view2():  
 return render\_template("c.html")

c.html file

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <title>CLIENT</title>  
</head>  
<body>  
 <h1>This is my Client Page</h1>  
</body>  
</html>

**Database in Flask**

To add database functionality to a Flask app, we will use SQLAlchemy.

SQLAlchemy is a Python SQL toolkit and object relational mapper (ORM) that enables Python to communicate with the SQL database system you prefer: MySQL, PostgreSQL, SQLite, and others. An ORM converts data between incompatible systems (object structure in Python, table structure in SQL database). SQLAlchemy is basically a bridge between Python and a SQL database.

Steps:

1.pip install mysqlclient.

2pip install flask\_sqlalchemy

In Flask, you can use the with statement to manage database connections and ensure they are properly opened and closed. The with statement is used in conjunction with a context manager, which allows you to define a setup and cleanup code that runs before and after a block of code. This ensures that the database connection is opened before processing the request and closed after the response is sent.

* Application Context: The application context represents the entire Flask application. It is created when a Flask application is being set up and is destroyed when the application is turn down. The application context is useful for storing global variables that should be accessible throughout the entire lifespan of the application.

To access the application context, you can use the current\_app object provided by Flask's app\_context() function, which allows you to access the current application instance within the context of a request.

* SQLALCHEMY\_DATABASE\_URI: The database URI to specify the database you want to establish a connection with. In this case, the URI follows the format.
* SQLALCHEMY\_TRACK\_MODIFICATIONS: A configuration to enable or disable tracking modifications of objects. You set it to False to disable tracking and use less memory. For more, see the configuration page in the Flask-SQLAlchemy documentation.

from flask import Flask,render\_template  
from flask\_sqlalchemy import SQLAlchemy  
  
app = Flask(\_\_name\_\_)  
  
with app.app\_context():#From Flask documentation: "Use app\_context() in a with block, and everything that runs in the block will have access to current\_app."  
 app.config["SQLALCHEMY\_DATABASE\_URI"]='mysql://root:root@localhost:3306/db1'   
 app.config['SQLALCHEMY\_TRACK\_MODIFICATION']=False  
  
 db = SQLAlchemy(app)  
  
 class laptop(db.Model):  
 lid = db.Column(db.Integer,primary\_key=True)  
 lnm = db.Column(db.String(34))  
 storage = db.Column(db.String(22))  
  
 db.create\_all()#table creation  
@app.route("/i")  
def insert():  
 ob = laptop(lid=1,lnm="hp")  
 db.session.add(ob)  
 db.session.commit()  
  
 return f"Added laptop {ob.lnm}"  
@app.route("/s")  
def show():  
 ob = laptop.query.all()  
 return render\_template("show.html",ob=ob)  
  
@app.route("/")  
def view():  
 return "database"  
  
  
if \_\_name\_\_ =="\_\_main\_\_":  
 app.run(debug=True)

db.create\_all() function does not recreate or update a table if it already exists. For example, if you modify your model by adding a new column, and run the db.create\_all() function, the change you make to the model will not be applied to the table if the table already exists in the database. The solution is to delete all existing database tables with the db.drop\_all() function and then recreate them with the db.create\_all() function like so:

**db.drop\_all()**

**db.create\_all()**

**CRUD IN FLASK**

from flask import Flask, render\_template, redirect,url\_for,request  
from flask\_sqlalchemy import SQLAlchemy  
  
  
app = Flask(\_\_name\_\_)  
  
with app.app\_context():  
 app.config["SQLALCHEMY\_DATABASE\_URI"] = "mysql://root:root@localhost:3306/b36"  
 app.config["SQLALCHEMY\_TRACk\_MODIFICATION"] = False  
  
 db = SQLAlchemy(app)  
 class Employee(db.Model):  
 eid = db.Column(db.Integer,primary\_key=True)  
 enm = db.Column(db.String(23))  
 esal = db.Column(db.Integer)  
 city = db.Column(db.String(34))  
  
 db.create\_all()  
  
  
@app.route("/")  
def view1():  
 return "database"  
  
@app.route("/v1",methods=["GET","POST"])  
def insert():  
 if request.method=="POST":  
 e = request.form["i"]  
 nm = request.form["n"]  
 sal = request.form["s"]  
 city = request.form["c"]  
 print(f"{e}---{nm}---{sal}")  
 ob = Employee(eid=e,enm=nm,esal=sal,city=city)  
 db.session.add(ob)  
 db.session.commit()  
 return redirect(url\_for("show"))  
 return render\_template("form.html")  
  
@app.route("/v2")  
def show():  
 ob = Employee.query.all()  
 return render\_template("show.html",ob=ob)  
  
@app.route("/v3/<int:eid>",methods=["GET","POST"])  
def update(eid):  
 ob = Employee.query.get(eid)  
 print(ob)  
 if request.method=="POST":  
 ob.eid = request.form["i"]  
 ob.enm = request.form["n"]  
 ob.esal = request.form["s"]  
 ob.city = request.form["c"]  
 db.session.commit()  
 return redirect(url\_for("show"))  
 return render\_template("update.html",ob=ob)  
  
@app.route("/v4/<int:eid>")  
def delete(eid):  
 obj = Employee.query.get(eid)  
 db.session.delete(obj)  
 db.session.commit()  
 return redirect(url\_for("show"))  
  
  
if \_\_name\_\_ =="\_\_main\_\_":  
 app.run(debug=True)

update.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
</head>  
<body>  
 <form method="post" action="/v3/{{ob.eid}}">  
 eid :<input type="number" name="i" value="{{ob.eid}}"><br><br>  
 enm :<input type="text" name="n" value="{{ob.enm}}"><br><br>  
 esal :<input type="text" name="s" value="{{ob.esal}}"><br><br>  
 city :<input type="text" name="c" value="{{ob.city}}"><br><br>  
 <input type="submit" value="UPDATE">  
 </form>  
  
</body>  
</html>

Show.html

<table border="2">  
 <tr>  
 <th>Emp Id</th>  
 <th>Emp Name</th>  
 <th>Emp Salary</th>  
 <th>City</th>  
 <th>Action</th>  
 </tr>  
  
 {% for i in ob %}  
 <tr>  
 <td>{{i.eid}}</td>  
 <td>{{i.enm}}</td>  
 <td>{{i.esal}}</td>  
 <td>{{i.city}}</td>  
  
 <td>  
<a href="/v3/{{i.eid}}"><button>UPDATE</button></a>  
<a href="/v4/{{i.eid}}"><button>DELETE</button></a>  
 </td>  
 {% endfor %}  
</table>

Form.html/add data

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
</head>  
<body>  
 <form method="post" >  
 eid :<input type="number" name="i"><br><br>  
 enm :<input type="text" name="n"><br><br>  
 esal :<input type="text" name="s"><br><br>  
 city :<input type="text" name="c"><br><br>  
 <input type="submit">  
 </form>  
  
</body>

**Authentication In Flask**

**pip uninstall Werkzeug**

**pip install Werkzeug==2.3.7**

**Steps:**

1.Install all library

1.pip install Flask.

2.pip install flask-sqlalchemy.

3.pip install flask-login.

2. Import the necessary modules

First import the classes we need from Flask, Flask-SQLAlchemy, and Flask-Login. We then create our flask application, indicate what database Flask-SQLAlchemy should connect to, and initialize the Flask-SQLAlchemy extension. We also need to specify a secret key, which can be any random string of characters, and is necessary as Flask-Login requires it to sign session cookies for protection again data tampering. Next, we initialize the LoginManager class from Flask-Login, to be able to log in and out users.

**from** flask **import** Flask

**from** flask\_sqlalchemy **import** SQLAlchemy

**from** flask\_login **import** LoginManager

# Create a flask application

app **=** Flask(\_\_name\_\_)

# Tells flask-sqlalchemy what database to connect to

app.config["SQLALCHEMY\_DATABASE\_URI"] **=** "[sqlite:///db.sqlite](sqlite://db.sqlite)"

# Enter a secret key

app.config["SECRET\_KEY"] **=** "ENTER YOUR SECRET KEY"

# Initialize flask-sqlalchemy extension

db **=** SQLAlchemy()

# LoginManager is needed for our application to be able to log in and out users

login\_manager **=** LoginManager()

login\_manager.init\_app(app)

3.Create a User Model & Database

To be able to store users’ information such as their username and password, we need to create a table with Flask-SQLAlchemy, this is done by creating a model that represents the information we want to store. In this case, we first create a Users class and make it a subclass of db.Model to make it a model with the help of Flask-SQLAlchemy. We also make the Users class a subclass of UserMixin, which will help to implement properties such as is\_authenticated to the Users class. We will also need to create columns within the user model, to store individual attributes, such as the user’s username. When creating a new column, we need to specify the datatype such as db.Integer and db.String as well. When creating columns, we also need to specify keywords such as unique = True, if we want to ensure values in the column are unique, nullable = False, which indicates that the column’s values cannot be NULL, and primary\_key = True, which indicates that the row can be identified by that primary\_key index. Next, the db.create\_all method is used to create the table schema in the database.

# Create user model

**class** Users(UserMixin, db.Model):

    id **=** db.Column(db.Integer, primary\_key**=**True)

    username **=** db.Column(db.String(250), unique**=**True,

                         nullable**=**False)

    password **=** db.Column(db.String(250),

                         nullable**=**False)

# Initialize app with extension

db.init\_app(app)

# Create database within app context

with app.app\_context():

    db.create\_all()

4.Adding a user loader

Before implementing the functionality for authenticating the user, we need to specify a function that Flask-Login can use to retrieve a user object given a user id. This functionality is already implemented by Flask-SQLAlchemy, we simply need to query and use the get method with the user id as the argument.

# Creates a user loader callback that returns the user object given an id

@login\_manager.user\_loader

**def** loader\_user(user\_id):

**return** Users.query.get(user\_id)

5.Registering new accounts with Flask-Login

Add the following code to a file name sign\_up.html in a folder called templates. To allow the user to register an account, we need to create the HTML. This will need to contain a form that allows the user to enter their details, such as their username and chosen password.

Signup.html

<h1>Signup-Form</h1>  
<form method="post" action="/su">  
 Username : <input type="text" name="u" required><br><br>  
 city : <input type="text" name="c" required><br><br>  
 Password : <input type="password" name="p" required><br><br>  
 <input type="submit" value="Register">  
  
</form>

Create a route that renders the template, and creates the user account if they make a POST request.

We create a new route with Flask by using the @app.route decorator. The @app.route decorator allows us to specify the route it accepts, and the methods it should accept. By default, it only accepts requests using the GET method, but when the form is submitted it is done using a POST request, so we’ll need to make POST an accepted method for the route as well. Within the register function that is called whenever the user visits that route, we can check if the method used was a POST request using the request variable that Flask provides and that needs to be imported. If a post request was made, this indicates the user is trying to register a new account, so we create a new user using the Users model, with the username and password set to whatever the user entered, which we can get by using request.form.get Lastly, we add the user object that was created to the session and commit the changes made. Once the user account has been created, we redirect them to a route with a callback function called “login”, which we will create in a moment. Ensure that you also import the redirect and url\_for functions from flask.

@app.route('/register', methods**=**["GET", "POST"])

**def** register():

  # If the user made a POST request, create a new user

**if** request.method **==** "POST":

        user **=** Users(username**=**request.form.get("username"),

                     password**=**request.form.get("password"))

        # Add the user to the database

        db.session.add(user)

        # Commit the changes made

        db.session.commit()

        # Once user account created, redirect them

        # to login route (created later on)

**return** redirect(url\_for("login"))

    # Renders sign\_up template if user made a GET request

**return** render\_template("sign\_up.html")

6. Allowing users to log in with Flask-Login

Like with creating the registered route, we first need a way for the user to log in through an HTML form. Add the following code to a file named login.html in the same templates folder.

Login.html

<h1>Login-Form</h1>  
<form method="post" action="/log">  
 Username : <input type="text" name="u"><br><br>  
 Password : <input type="password" name="p"><br><br>  
 <input type="submit" value="Login">  
  
</form>

Add the functionality to log in to the user within a login function for the /login route.

With the login route, we do the same thing of checking if the user made a POST request. If they did, we filter the users within the database for a user with the same username as the one being submitted. Next, we check if that user has the same password as the password the user entered in the form. If they are the same, we log-in to the user by using the login\_user function provided by Flask-Login. We can then redirect the user back to a route with a function called “home”, which we will create in a moment. If the user didn’t make a POST request, and instead a GET request, then we’ll render the login template.

@app.route("/login", methods**=**["GET", "POST"])

**def** login():

    # If a post request was made, find the user by

    # filtering for the username

**if** request.method **==** "POST":

        user **=** Users.query.filter\_by(

            username**=**request.form.get("username")).first()

        # Check if the password entered is the

        # same as the user's password

**if** user.password **==** request.form.get("password"):

            # Use the login\_user method to log in the user

            login\_user(user)

**return** redirect(url\_for("home"))

        # Redirect the user back to the home

        # (we'll create the home route in a moment)

**return** render\_template("login.html")

7. Conditionally rendering HTML based on the user’s authentication status with Flask-Login

When using Flask, it uses Jinja to parse the templates. Jinja is a templating engine that allows us to add code, such as if-else statements within our HTML, we can then use it to conditionally render certain elements depending on the user’s authentication status for example the current\_user variable is exported by Flask-Login, and we can use it within the Jinja template to conditionally render HTML based on the user’s authentication status.

final.html

<body>

<h1>Final</h1>

{% if current\_user.is\_authenticated %}  
 <h1>hello {{current\_user.username}}!!!</h1>  
 {% endif %}  
 <h2>You have successfully Logged In!!!</h2>  
  
<a href="{{url\_for('logout')}}">Logout</a>  
</body>

8. Adding Logout Functionality

Here, we will update the home.html template to the following to add a logout link, and this will give the homepage a link to log out the user if they are currently logged in.

from flask import Flask,render\_template,request,redirect,url\_for  
from flask\_sqlalchemy import SQLAlchemy  
from flask\_login import LoginManager,UserMixin,login\_user,logout\_user,login\_required  
  
  
app = Flask(\_\_name\_\_)  
  
with app.app\_context():  
 app.config["SQLALCHEMY\_DATABASE\_URI"] = "mysql://root:root@localhost:3306/b37"  
 app.config["SQLALCHEMY\_TRACK\_MODIFICATIONS"] = False  
 app.config["SECRET\_KEY"] = "jhgafet54try"  
  
 db = SQLAlchemy(app)  
  
 login\_manager = LoginManager()#object is used to hold all settings used for logging  
 login\_manager.init\_app(app)#to configure application for login  
 login\_manager.login\_view = "login" #loginview name  
  
 class User(db.Model,UserMixin):  
 id = db.Column(db.Integer,primary\_key=True)  
 username = db.Column(db.String(34),nullable=False)  
 password = db.Column(db.String(45),nullable=False)  
 db.create\_all()  
  
  
  
  
  
@app.route("/")  
def database():  
 return "Database Created"  
  
@app.route("/h")  
def home():  
 return render\_template("home.html")  
  
@login\_manager.user\_loader  
def load\_user(user\_id):  
 return User.query.filter\_by(id = user\_id).first()

# fetch the user\_id in user table if user\_id is present it return object of specified userid if it is not present it return None  
  
@app.route("/su",methods=["GET","POST"])  
def signup():  
 if request.method == "POST":  
 unm = request.form["u"]  
 pwd = request.form["p"]  
 obj = User(username=unm, password=pwd)  
 db.session.add(obj)  
 db.session.commit()  
 return redirect(url\_for("login"))  
 return render\_template("signup.html")  
  
  
@app.route("/log" ,methods=["GET","POST"])  
def login():  
 if request.method=="POST":  
 u = request.form.get("u")  
 p = request.form.get("p")  
 user = User.query.filter\_by(username=u,password=p).first()

#load\_user function will get call  
 if user:  
 login\_user(user)  
 return redirect(url\_for("final"))  
 return render\_template("login.html")  
  
  
@app.route("/final")  
@login\_required  
def final():  
 return render\_template("final.html")  
  
@app.route("/logout")  
@login\_required  
def logout():  
 logout\_user()  
 return redirect(url\_for("login"))  
  
  
if \_\_name\_\_ =="\_\_main\_\_":  
 app.run(debug=True)

Home.html

<h1>Home Page</h1>  
<a href="{{url\_for('signup')}}">Signup</a><br><br>  
<a href="{{url\_for('login')}}">Login</a><br><br>

**Error Handling in Flask**

Error handling in Flask is a crucial aspect of building web applications to handle unexpected situations and provide meaningful responses to users. Flask provides several mechanisms to handle errors gracefully. Here are some common techniques for error handling in Flask:

Default Error Handling: By default, Flask already handles some common errors like 404 (Not Found) and 500 (Internal Server Error) and displays a basic error page. However, you can customize these error pages to provide a more user-friendly experience.

Custom Error Pages: Flask allows you to create custom error pages for specific HTTP error codes. To do this, you can use the errorhandler decorator from the Flask app object. For example:

from flask import Flask, render\_template

app = Flask(\_\_name\_\_)

@app.errorhandler(404)

def page\_not\_found(error):

return render\_template('404.html'), 404

@app.errorhandler(500)

def internal\_server\_error(error):

return render\_template('500.html'), 500

In this example, when a 404 or 500 error occurs, Flask will invoke the respective error handler functions, and the corresponding custom HTML templates will be rendered.

Global Error Handling: To handle all types of errors in one central location, you can use the @app.errorhandler decorator without specifying any error code. This becomes a global error handler that will handle any uncaught exceptions in your application:

from flask import Flask, render\_template

app = Flask(\_\_name\_\_)

@app.errorhandler(Exception)

def handle\_exception(error):

# Log the error or perform any necessary actions

return render\_template('error.html', error=error), 500

Abort Function: Flask provides an abort() function that allows you to trigger an error with a specific HTTP status code. For instance:

from flask import Flask, abort

app = Flask(\_\_name\_\_)

@app.route('/example/<int:value>')

def example(value):

if value < 0:

abort(400, "Invalid value. Must be greater than 0.")

return f"Value: {value}"

In this case, when a negative value is passed, Flask will return a 400 Bad Request error with the custom message.