Contents

[FLASK 2](#_Toc186104475)

[VIRTUAL ENVIRONMENT 2](#_Toc186104476)

[SETTING UP VENV IN WINDOWS (USING PIP) 2](#_Toc186104477)

[SETTING UP FLASK 3](#_Toc186104478)

[ROUTES 3](#_Toc186104479)

[DYNAMIC ROUTING 4](#_Toc186104480)

[DEBUG MODE 4](#_Toc186104481)

[EXAMPLE 4](#_Toc186104482)

[TEMPLATES 6](#_Toc186104483)

[TEMPLATING ENGINE 6](#_Toc186104484)

[TEMPLATE VARIABLE 7](#_Toc186104485)

[TEMPLATE INHERITANCE 9](#_Toc186104486)

[HELPER FUNCTIONS 9](#_Toc186104487)

[url\_for() 10](#_Toc186104488)

[404 11](#_Toc186104489)

[TEMPLATE FORMS 11](#_Toc186104490)

[EXAMPLE 12](#_Toc186104491)

[FORMS IN FLASK 13](#_Toc186104492)

[DATABASE – SQLALCHEMY 14](#_Toc186104493)

[ORM 14](#_Toc186104494)

[SETTING UP THE ORM 15](#_Toc186104495)

[CRUD OPERATION 15](#_Toc186104496)

[REST API 15](#_Toc186104497)

[CRUD APPLICATION 16](#_Toc186104498)

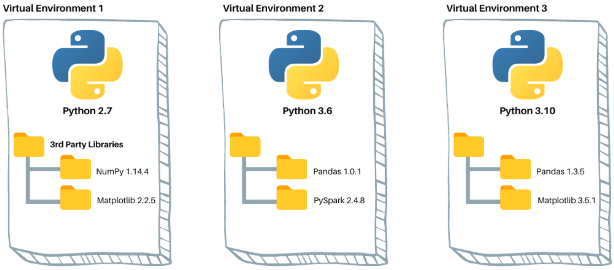
[DEPLOYMENT – HEROKU 16](#_Toc186104499)

# FLASK

* Flask is a web framework for building web applications in Python.
* It is known for its simplicity and flexibility, making it a great choice for both small and large-scale projects.
* Flask follows the model-view-controller (MVC) architectural pattern and provides a lightweight and modular approach to web development.
* Documentation: [Welcome to Flask — Flask Documentation (3.1.x)](https://flask.palletsprojects.com/en/stable/)

## VIRTUAL ENVIRONMENT

* A virtual environment in Python is an isolated environment that allows us to have separate Python installations and package dependencies for different projects.
* It helps in managing and organizing project-specific dependencies, ensuring that each project has its own set of required packages without interfering with other projects or the system-wide Python installation.
* In a virtual environment, we can install specific versions of Python packages and maintain consistency across different projects.
* By isolating dependencies, we can avoid conflicts that may arise when different projects require different versions of the same package.



### SETTING UP VENV IN WINDOWS (USING PIP)

Step 1: Create a directory where the virtual environment needs to be set up

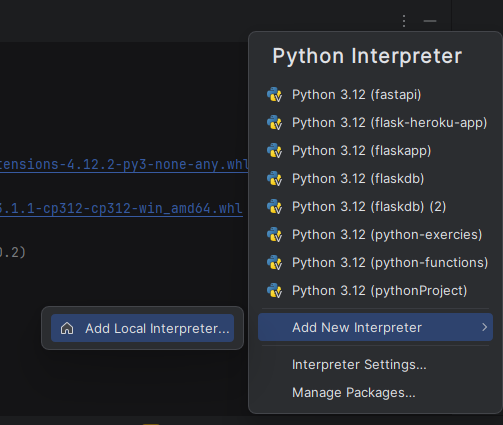
Step 2: Run the following command in the cmd line

|  |  |
| --- | --- |
| CREATE THE VIRTUAL ENVIRONMENT:   * Virtual environments are created using tools like **virtualenv** or the built-in **venv module** in Python. * These tools create a separate directory with its own Python executable, libraries, and scripts. | python -m venv <**virtual\_env\_name**>  python -m venv flaskenv |
| ACTIVATE THE VIRTUAL ENVIRONMENT: | <**virtual\_env\_name**>\Scripts\activate.bat  flaskenv\Scripts\activate.bat |
|  | |
| * Once inside a virtual environment, we can use the Python interpreter and install packages using tools like pip. * Any packages installed in the virtual environment will only be available within that environment and won't interfere with other projects or the system. | |
| For example – Let install the “flask” package in the virtual environment (fastapiapp)  **pip install flask**   * When we execute – **pip list** command it in the virtual environment it will show the flask in installed packages list |  |

***NOTE: -When we activate a virtual environment, it modifies your system's environment variables to prioritize the isolated environment over the system-wide Python installation.***

#### CONFIGURE VIRTUAL ENVIRONMENT IN PYCHARM

* Click on the interpreter (right buttom corner in the IDE) 🡪 Add New Interpreter🡪 Add local Interpreter



A screen shot of a computer

Description automatically generated

## SETTING UP FLASK

|  |  |
| --- | --- |
| INSTALL FLASK | **pip install flask** |
| RUNNING FLASK APP | **flask run** |

## ROUTES

|  |  |
| --- | --- |
| from flask import Flask app = Flask(\_\_name\_\_)  @app.get('/') def index():  return "Hello, World!"  if \_\_name\_\_ == '\_\_main\_\_':  app.run(port=5000) | **BASIC FLASK WEB APPLICATION**   * `**from flask import Flask**`: This line imports the Flask module, which is a Python framework for building web applications. * **`app = Flask(\_\_name\_\_)`:** This line creates a Flask application instance. The `\_\_name\_\_` variable is a special Python variable that represents the name of the current module. When this code is run as the main module, `\_\_name\_\_` will be set to `\_\_main\_\_`. * **`@app.get('/')`:** This is a decorator that associates the following function with the specified URL route. In this case, the function `index()` is associated with the root URL `/`. * **`if \_\_name\_\_ == '\_\_main\_\_**':`: This conditional statement checks if the script is being run directly (as the main module). If it is, the following code block will be executed. |

### DYNAMIC ROUTING

|  |  |
| --- | --- |
| @app.get('/profile**/<profileId>**') def userProfile(profileId):  return "User Profile Page of {}".format(profileId) | `@app.route('/profile/<profileId>')`:   * This is a decorator that associates the following function with a URL route pattern `/profile/<profileId>`. * The `<profileId>` part is a variable placeholder that captures the value provided in the URL and passes it as an argument to the function. * By using the `<profileId>` placeholder in the URL route, the code allows for dynamic profile pages. When a user visits a URL like `/profile/123`, the `profileId` argument in the `userProfile()` function will be set to `123`, and the function will return a response with the corresponding user profile information. |

## EXAMPLE – REST API

We will set up a simple REST API that allows us to:

* Create stores, each with a name and a list of stocked items.
* Create an item within a store, each with a name and a price.
* Retrieve a list of all stores and their items.
* Given its name, retrieve an individual store and all its items.
* Given a store name, retrieve only a list of item within it.

|  |  |  |
| --- | --- | --- |
| **ACTION** | **REQUEST** | **RESPONSE** |
| **CREATE STORES** | **`POST /store {"name": "My Store"}** | {"name": "My Store", "items": []} |
| **CREATE ITEMS** | **POST /store/My Store/item {"name": "Chair", "price": 175.50}** | {"name": "Chair", "price": 175.50} |
| **RETRIEVE ALL STORES AND THEIR ITEMS** | **GET /store** | {  "stores": [  {  "name": "My Store",  "items": [  {  "name": "Chair",  "price": 175.50  }  ]  }  ] } |
| **GET A PARTICULAR STORE** | **GET /store/My Store** | {  "name": "My Store",  "items": [  {  "name": "Chair",  "price": 175.50  }  ] } |
| **GET ONLY ITEMS IN A STORE** | **GET /store/My Store/item** | [  {  "name": "Chair",  "price": 175.50  } ] |

### CODE

|  |
| --- |
| from flask import Flask, request  app = Flask(\_\_name\_\_)  stores = [  {  "name": "My Store",  "items": [  {  "name": "Chair",  "price": 15.99  }  ]  }  ]  @app.get("/store") # http://127.0.0.1:5000/store  def get\_stores():  return {"stores": stores}  @app.post("/store")  def create\_store():  request\_data = request.get\_json()  new\_store = {"name": request\_data["name"], "items": []}  stores.append(new\_store)  return new\_store, 201  @app.post("/store/<string:name>/item")  def create\_item(name):  request\_data = request.get\_json()  for store in stores:  if store["name"] == name:  new\_item = {"name": request\_data["name"], "price": request\_data["price"]}  store["items"].append(new\_item)  return new\_item, 201  return {"message": "Store not found"}, 404  @app.get("/store/<string:name>")  def get\_store(name):  for store in stores:  if store["name"] == name:  return store  return {"message": "Store not found"}, 404  @app.get("/store/<string:name>/item")  def get\_item\_in\_store(name):  for store in stores:  if store["name"] == name:  return {"items": store["items"]}  return {"message": "Store not found"}, 404 |

### RUNING THE APPLICATION IN DOCKER

## requirements.txt

**A `requirements.txt` file is commonly used in Python projects to specify the dependencies and their versions required for the project to run. It is useful for managing and reproducing the project's environment and avoid compatibility issues (like package.json)**

### STEPS TO CREATE requirements.txt

* **Create a `requirements.txt` file**: In the project directory, create a new file named **requirements.txt**

|  |  |
| --- | --- |
| * **Add dependencies**:   1. In the `requirements.txt` file, list the required dependencies, each on a separate line.   2. Each line typically consists of the package name and version specification. | Flask==2.1.0  requests>=2.0.0 |
| * **Specify versions**   1. Specify specific versions of packages using the equality operator (`==`), or you can use comparison operators (`>=`, `<=`, `>`, `<`) to specify version ranges.   2. We can also omit the version specification to allow any compatible version. |  |
| * **Install dependencies**:   1. To install the project dependencies from the `requirements.txt` file, use the following command:   2. This command will read the `requirements.txt` file and install the specified packages and their versions into your Python environment | * pip install -r requirements.txt |
| * **Updating dependencies:**   1. If we add or modify dependencies in the `requirements.txt` file, we can update our environment by running the same `**pip install -r requirements.txt**` command again. |  |

#### requirements.txt FOR THE “STORE” PROJECT

Create and install dependencies using requirement.txt

**STEP 1**: Create requirements.txt

|  |  |
| --- | --- |
| **flask flask-smorest python-dotenv** |  |
| **STEP 2**: Install the dependencies using requirements.txt (mentioned in requirements.txt)  **pip install -r requirements.txt** | |

## SETTING ENVIRONMENT VARIABLES

* The **.flaskenv** file is a text file that is commonly used in Flask projects to store environment variables.
* It is used to define and manage environment-specific configurations for your Flask application.

|  |  |
| --- | --- |
| * The .flaskenv file is typically placed in the root directory of the Flask project and follows the syntax of key-value pairs, where each line represents an environment variable declaration. * The .flaskenv file should not be committed to version control systems, as it may contain sensitive information like secret keys or API tokens. * It is recommended to add the file to the .gitignore or equivalent file to prevent accidental exposure of sensitive data. |  |

In the above example - The environment variables `**FLASK\_APP**` and `**FLASK\_DEBUG**` are commonly used in Flask applications to configure the Flask development server. Here's what they do:

1. **FLASK\_APP**:
   1. This environment variable is **used to specify the name of the Flask application** or **the file containing the Flask application object**. By setting `FLASK\_APP` to the name or path of your Flask application file, we inform Flask about which application to run. For example, if your Flask application is defined in a file named `app.py`, we can set `FLASK\_APP` as follows:

**Windows: `set FLASK\_APP=app.py`**

**Mac/Linux: `export FLASK\_APP=app.py`**

1. **FLASK\_DEBUG**:
   1. This environment variable is used to **enable the Flask development server's debug mode**.
   2. When `**FLASK\_DEBUG` is set to `1`, Flask will automatically reload the server whenever changes are made to the application code.**
   3. It also provides more detailed error messages in the browser when an error occurs. To enable debug mode, set `FLASK\_DEBUG` as follows:

**Windows: `set FLASK\_DEBUG=1`**

**Mac/Linux: `export FLASK\_DEBUG=1`**

* After setting these environment variables, we can use the `**flask run**` command to start the Flask development server with the specified application and debug mode. **For example, if we set `FLASK\_APP` to `app.py` and `FLASK\_DEBUG` to `1`, we can start the server by running `flask run`.**

*Note: When running the Flask application in a production environment, it is recommended to disable the debug mode by either removing the `FLASK\_DEBUG` environment variable or setting it to `0`. Debug mode should only be enabled during development, as it may expose sensitive information and impact performance.*

## REST API USING SMOREST

|  |  |
| --- | --- |
| **Creating a database file** | Create a db.py file with the following content:  db.py  stores = {} items = {} |
| **IMPORT “**stores, items**” in app.py** | from db import stores, items |

### DATA MODELING UPDATES

* Documentation : <https://rest-apis-flask.teclado.com/docs/flask_smorest/data_model_improvements/>

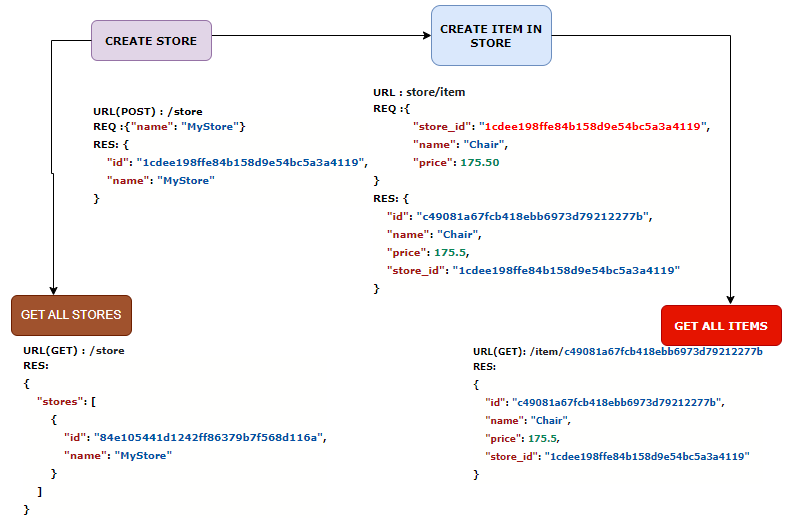
A computer code with text and numbers

Description automatically generated with medium confidence

**TABLE VIEW**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| store | |  | items | | | |
| store\_id (PK) | name |  | id | name | price | store\_id (FK) |

|  |  |
| --- | --- |
| OLD VIEW METHOD | UPDATED VIEW METHOD |
| **get\_store : GET STORE BASED IN STORE ID** | |
| @app.get("/store/<string:name>")  def get\_store(name):  for store in stores:  if store["name"] == name:  return store  return {"message": "Store not found"}, 404 | @app.get("/store/<string:**store\_id**>")  def get\_store(store\_id):  try:  # Here you might also want to add the items in this store  # We'll do that later on in the course  return stores[store\_id]  except KeyError:  return {"message": "Store not found"}, 404 |
| **get\_stores : GET ALL STORES** | |
| @app.get("/store")  def get\_stores():  return {"stores": stores} | @app.get("/store")  def get\_stores():  return {"stores": list(stores.values())} |
| **CREATE STORE** | |
| @app.post("/store")  def create\_store():  request\_data = request.get\_json()  new\_store = {"name": request\_data["name"], "items": []}  stores.append(new\_store)  return new\_store, 201 | import uuid  @app.post("/store")  def create\_store():  store\_data = request.get\_json()  store\_id = uuid.uuid4().hex  store = {\*\*store\_data, "id": store\_id}  stores[store\_id] = store  return store |
| **CREATE ITEM** | |
| @app.post("/store/<string:name>/item")  def create\_item(name):  request\_data = request.get\_json()  for store in stores:  if store["name"] == name:  new\_item = {"name": request\_data["name"], "price": request\_data["price"]}  store["items"].append(new\_item)  return new\_item, 201  return {"message": "Store not found"}, 404 | @app.post("/item")  def create\_item():  item\_data = request.get\_json()  if item\_data["store\_id"] not in stores:  return {"message": "Store not found"}, 404  item\_id = uuid.uuid4().hex  item = {\*\*item\_data, "id": item\_id}  items[item\_id] = item  return item |
| **GET ALL ITEMS** | |
| @app.get("/item")  def get\_all\_items():  return {"items": list(items.values())} |  |
| **GET ITEM BY ITEM ID** | |
| @app.get("/store/<string:name>/item")  def get\_item\_in\_store(name):  for store in stores:  if store["name"] == name:  return {"items": store["items"]}  return {"message": "Store not found"}, 404 | @app.get("/item/<string:item\_id>")  def get\_item(item\_id):  try:  return items[item\_id]  except KeyError:  return {"message": "Item not found"}, 404 |



#### MORE ENDPOINTS

* DELETE /item/<string:item\_id> so we can delete items from the database.
* PUT /item/<string:item\_id> so we can update items.
* DELETE /store/<string:store\_id> so we can delete stores.

|  |  |
| --- | --- |
| **DELETE STORE**  @app.delete('/store/<string:store\_id>')  def delete\_store(store\_id):  try:  del stores[store\_id]  return {'message': 'Store deleted'}, 200  except KeyError:  return {'message': 'Store not found'}, 404 | **DELETE ITEM**  @app.delete('/item/<string:item\_id>')  def delete\_item(item\_id):  try:  del items[item\_id]  return {'message': 'Item deleted'}, 200  except KeyError:  return {'message': 'Item not found'}, 404 |
| **UPDATE ITEM**  @app.put('/store/<string:item\_id>')  def update\_item(item\_id):  item\_data = request.get\_json()  if item\_id not in items:  return {'message': 'Item not found'}, 404  items[item\_id] = item\_data  return items[item\_id], 200 |  |

## DEBUG MODE

|  |  |
| --- | --- |
| from flask import Flask app = Flask(\_\_name\_\_)  @app.route('/') def index():  return "Hello, World!"  if \_\_name\_\_ == '\_\_main\_\_':  app.run(port=5000, **debug=True)** | from flask import Flask  app = Flask(\_\_name\_\_)    # Enable debug mode  app.debug = True    @app.route('/')  def index():  return "Hello, World!"    if \_\_name\_\_ == '\_\_main\_\_':  app.run(port=5000) |

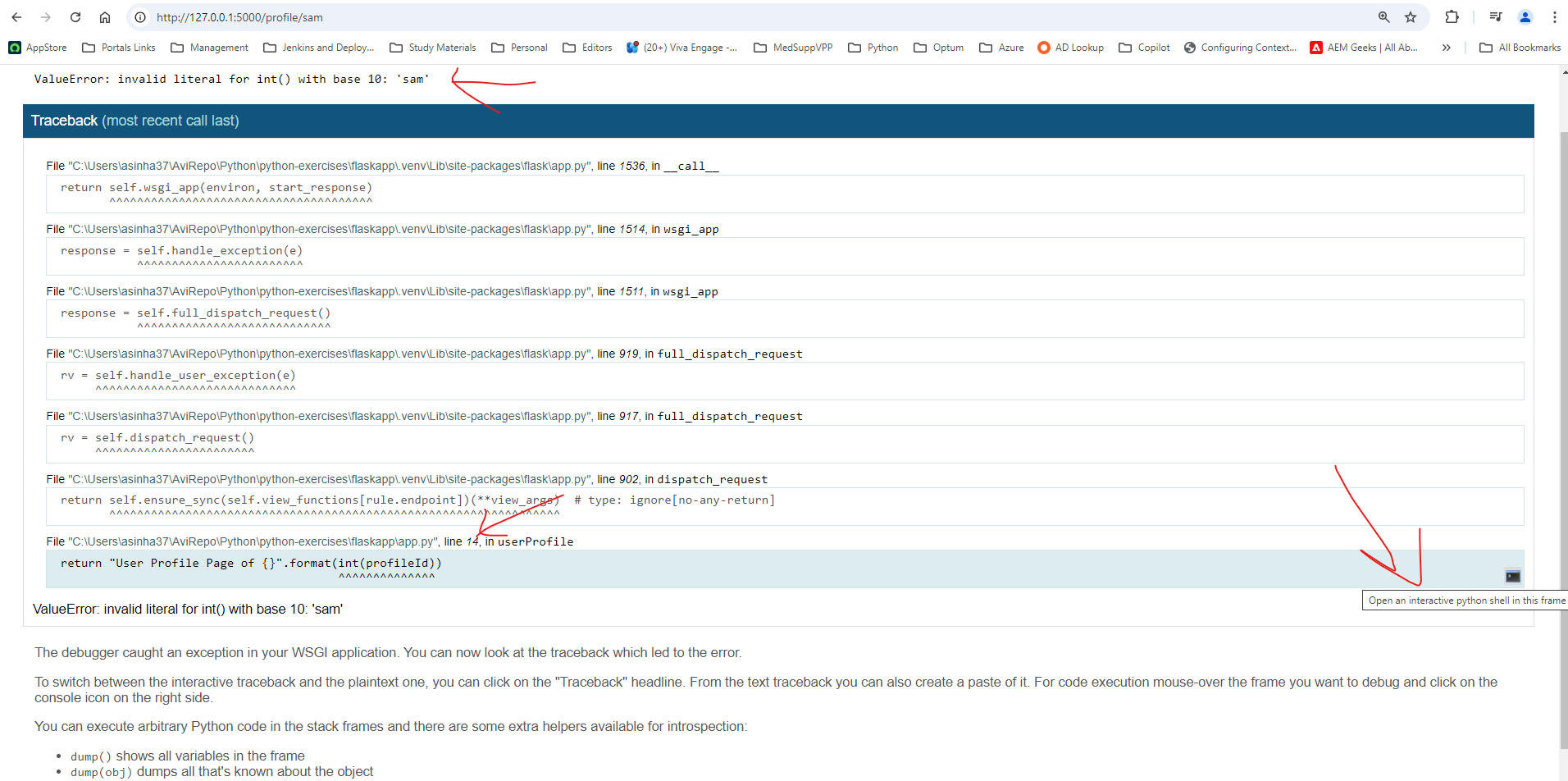
When debug mode is enabled, the following features are available:

* **Detailed error messages**:
  + If an error occurs in the code, Flask will display a detailed traceback with information about where the error occurred, making it easier to identify and fix issues.
* **Automatic reloading**:
  + When a change is made to the code while the application is running, Flask automatically detects the change and reloads the application.
  + This eliminates the need to manually stop and restart the server after each code modification.

### EXAMPLE

|  |  |
| --- | --- |
| @app.route('/profile/<profileId>') def userProfile(profileId):  return "User Profile Page of {}".format(int(profileId)) | Let say we have following code in flask route. The logic will break for request like , due to type casting  <http://127.0.0.1:5000/profile/sam> |

* If debug mode is ON , python will provide the detailed stack trace of the error and a python terminal which can be accessed using Debugger PIN
* Click on the terminal Icon which will prompt for the debugger PIN



**DEBUGGER PIN IN TERMINAL**

A screen shot of a computer

Description automatically generated

* Once the console is enabled – we can check the code and values of variables to debug the issue

A screenshot of a computer program

Description automatically generated

## TEMPLATES

* Templates are the view which is render for a given route

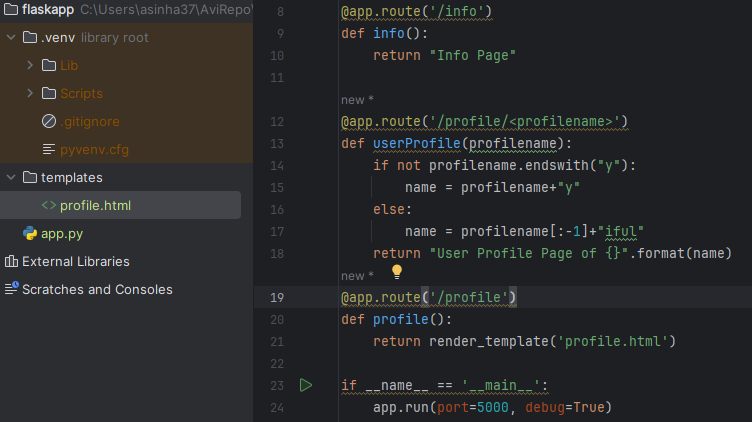
### TEMPLATING ENGINE

* **Jinja2 is the default templating engine used by Flask, a popular Python web framework. Flask integrates Jinja2 seamlessly to render dynamic content in web applications.**

**TO USE JINJA2 IN FLASK, YOU NEED TO FOLLOW THESE STEPS:**

|  |  |
| --- | --- |
| **IMPORT THE NECESSARY MODULES** | * In Flask application file, import the `render\_template` function from the `flask` module. * This function will be used to render Jinja2 templates. |
| **CREATE A TEMPLATES FOLDER** | * In the project directory, create a folder named "templates". * Flask expects your Jinja2 template files to be stored in this folder. |
| **CREATE A JINJA2 TEMPLATE** | * Inside the "templates" folder, create a Jinja2 template file with a `.html` extension. * For example, let's create a file named "index.html" as an example template. |
| **DEFINE A ROUTE IN YOUR FLASK APPLICATION** | * In the Flask application file, define a route for the URL where we want to render the Jinja2 template. For example, let's define a route for the root URL ("/")   app = Flask(\_\_name\_\_)  @app.route('/')  def index():  return render\_template('index.html')  if \_\_name\_\_ == '\_\_main\_\_':  app.run() |
| Use Jinja2 syntax in your template | * Open the Jinja2 template file ("index.html" in this case) and use Jinja2 syntax to define variables, control structures, and other template-related logic. Jinja2 uses double curly braces {{ }} for variables and control structures like `if`, `for`, etc * Example   **<!-- index.html -->**  **<!DOCTYPE html>**  **<html>**  **<head>**  **<title>My Flask App</title>**  **</head>**  **<body>**  **<h1>Hello, {{ name }}!</h1>**  **</body>**  **</html** |

#### SETTING UP TEMPLATE



**To set up a template using render\_template() in Flask, you need to follow these steps**:

1. **Create a "templates" folder**: In the project directory, create a folder named "**templates**". Flask expects the template files to be stored in this folder.
2. **Create a template file**: Inside the "templates" folder, create an HTML file for your template. For example, let's create a file called "index.html" as an example.
3. **Define a route in your Flask application**: In your Flask application file, define a route for the URL where you want to render the template. For example, let's define a route for the root URL ("/"):
4. **Render the template**: Inside the route function, use the **render\_template()** function to render the template. Pass the name of the template file as an argument to the function. For example, in the code above, we are rendering the "index.html" template.
5. **Customize the template**: Open the template file ("index.html" in this case) and customize it based on your needs. You can use HTML, CSS, and Jinja2 templating syntax to create the structure and content of your template. You can also include variables, control structures, and template inheritance as needed.

### TEMPLATE VARIABLE

* We can pass variables from from Flask views to to templates using the `**render\_template()`** function.
* Inside the template (HTML), we can access these variables using double curly braces **{{ variable\_name }}.** For example, if you pass a variable named `name` to the template, you can display it as **{{ name }}.**

|  |  |
| --- | --- |
| ROUTE | TEMPLATE(HTML) |
| @app.route('/profile') def profile():  pageMetaData = {  "title": "Profile Page",  "description": "This is the profile page of the user"  }   profileDetails= {  "name": "John Doe",  "age": 25,  "email": "avishekh.sinha@gmail.com"  }  name = profileDetails["name"]  return render\_template('profile.html', profileDetails=profileDetails, pageMetaData=pageMetaData) | <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8"> <h1>{{pageMetaData["title"]}}</h1>  </head> <body> <h1>Profile Details</h1> <table border="1">  <tr>  <th>Key</th>  <th>Value</th>  </tr>  **{% for key, value in profileDetails.items() %}  <tr>  <td>{{ key }}</td>  <td>{{ value }}</td>  </tr>  {% endfor %}** </table> </body> </html> </body> </html> |

#### TEMPLATE FILTER

#### CONTROL FLOW

|  |  |
| --- | --- |
| ROUTE | TEMPLATE(HTML) |
|  |  |
| @app.route('/profile') def profile():  pageMetaData = {  "title": "Profile Page",  "description": "This is the profile page of the user"  }   profiles = [  {  "name": "John Doe",  "age": 25,  "email": "john.doe@example.com",  "hobbies": ["reading", "travelling", "coding"]  },  {  "name": "Jane Smith",  "age": 30,  "email": "jane.smith@example.com",  "hobbies": ["painting", "hiking", "music"]  }  ]   return render\_template('profile.html', profiles=profiles, pageMetaData=pageMetaData) | <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>Profile Details</title> </head> <body> <h1>{{pageMetaData["title"]}}</h1> <table border="1">  <tr>  <th>Key</th>  <th>Value</th>  </tr>  {% for profile in profiles %}  {% for key, value in profile.items() %}   <tr>  <td>{{ key }}</td>  <td>  {% if key == "age" and value| int > 25 %}  <span style="color:red">{{ value }}</span>  {% else %}  <span>{{ value }}</span>  {% endif %}  </td>  </tr>  {% endfor %}  {% endfor %} </table> </body> </html> </body> </html> |

### TEMPLATE INHERITANCE

* Template inheritance in Flask allows us to create a base template with common elements (e.g., header, footer) and extend it in other templates. This helps in reducing code duplication and maintaining consistency across multiple pages.

#### EXAMPLE

|  |  |
| --- | --- |
| **STEP 1: CREATE A BASE TEMPLATE**:   1. Create a base template that will serve as the foundation for other templates. 2. Let's call it "base.html". It should contain the common HTML structure and any elements that will be shared across multiple pages | **<!-- base.html -->**  <!DOCTYPE html>  <html>  <head>  <title>{% block title %}My Website{% endblock %}</title>  </head>  <body>  <header>  <!-- Common header content -->  </header>  <main>  {% block content %}{% endblock %}  </main>    <footer>  <!-- Common footer content -->  </footer>  </body>  </html> |
| **EXTEND THE BASE TEMPLATE IN OTHER TEMPLATES**:   * In other templates, we can extend the base template using the `{% extends %}` directive. * Within the child templates, we can override specific blocks defined in the base template using the `{% block %}` directive. For example: | <!-- home.html -->  {% extends 'base.html' %}  {% block title %}Home - My Website{% endblock %}    {% block content %}  <!-- Home page specific content -->  {% endblock %} |
|  | <!-- about.html -->  {% extends 'base.html' %}  {% block title %}About - My Website{% endblock %}  {% block content %}  <!-- About page specific content -->  {% endblock %} |

## HELPER FUNCTIONS

### url\_for()

A screenshot of a computer

Description automatically generated

* **`url\_for()` is used to generate URLs for specific routes in the application.**
* The `url\_for()` function takes the name of a route as its first argument and optional keyword arguments representing the variable parts of the route. It then returns the URL for that route.

|  |  |  |
| --- | --- | --- |
| **App.py** | **Home.html(http://localhost:5000/home)** | **Info.html** |
| from flask import Flask, render\_template,request app = Flask(\_\_name\_\_)  @app.route('/home') def **index**():  return render\_template('home.html')  @app.route('/info') def **info**():  username = request.args.get('username')  return render\_template('info.html')  if \_\_name\_\_ == '\_\_main\_\_':  app.run(port=5000, debug=True) | **<a href="{{url\_for('info',username='john')}}">Profile Page</a>** | URL : <http://localhost:5000/info?username=john>  <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>Information</title> </head> <body>  **User Name request.args.get('username')}}** </body> </html> |

#### STATIC FILE

|  |  |
| --- | --- |
|  | <html> <head>  <title>{% block title %}My amazing site{% endblock %}</title>  <**link rel="stylesheet" href="{{url\_for('static', filename='css/bootstrap.min.css')}}">** </head> <body> <div class="container">  <nav class="navbar bg-dark border-bottom border-body" data-bs-theme="dark"> …. </nav>  <div class="row">  {% block content %}{% endblock %}  </div> </div> **<script src="{{url\_for('static',filename='js/bootstrap.bundle.min.js')}}"></script>** </body> </html> |

### 404

To create a custom 404 error page to handle requests for routes that do not exist. Flask provides a decorator **`@app.errorhandler**` that allows us to define a function to handle specific error codes.

|  |  |
| --- | --- |
| **app.py**  from flask import Flask, render\_template    app = Flask(\_\_name\_\_)    **@app.errorhandler(404)**  def page\_not\_found(e):  return render\_template('404.html'), 404    if \_\_name\_\_ == '\_\_main\_\_':  app.run() | **404.html**  <!DOCTYPE html>  <html>  <head>  <title>404 Not Found</title>  </head>  <body>  <h1>404 Not Found</h1>  <p>The requested page does not exist.</p>  </body>  </html> |

* In the above example, we define a function `page\_not\_found` to handle the 404 error. Inside the function, we use `render\_template()` to render a custom template `404.html`. **We also specify the HTTP status code `404` as the second parameter to `render\_template()`.**
* The `@app.errorhandler(404)` decorator binds the `page\_not\_found` function to handle any 404 errors that occur in the application.
* Note that the `@app.errorhandler` decorator can be used for handling other error codes as well, such as 500 for internal server errors.

|  |  |
| --- | --- |
|  |  |

## TEMPLATE FORMS



|  |  |
| --- | --- |
| **home.html**  **<a href="{{url\_for('info',username='john')}}">Profile Page</a>  <a href="{{url\_for('signup')}}">Sign Up</a>** | **app.py**  from flask import Flask, render\_template,request app = Flask(\_\_name\_\_)  @app.route('/home') def index():  return render\_template('home.html')  @app.route('/signup') def signup():  return render\_template('signup.html')  @app.route('/thankyou') def thankyou():  firstname = request.args.get('first\_name')  lastname = request.args.get('last\_name')  full\_name = firstname + " " + lastname  print(full\_name)  return render\_template('thank\_you.html', full\_name=full\_name)  if \_\_name\_\_ == '\_\_main\_\_':  app.run(port=5000, debug=True) |
| **signup.html**  <form action="{{url\_for('thankyou')}}">  <label for="first\_name">First Name:</label>  <input type="text" id="first\_name" name="first\_name" required><br><br>  <label for="last\_name">Last Name:</label>  <input type="text" id="last\_name" name="last\_name" required><br><br>  <input type="submit" value="Submit">  </form> | **thank\_you.html**  <h1>Thank You! {{full\_name}}</h1> <p>Your submission has been received.</p> <p><a href="{{url\_for('index')}}">Go to Home</a></p> |

## EXAMPLE

Create a form which will accept user name with following criteria

* Must have uppercase
* Must have lowercase
* Ends with number

|  |
| --- |
| **index.html**  <form action="{{url\_for('thankyou')}}">  <div class="mb-3">  **<label for="user\_name" class="form-label">User Name</label>**  **<input type="text" class="form-control" id="user\_name" aria-describedby="emailHelp" name="user\_name">**  </div>  <button type="submit" class="btn btn-primary">Submit</button>  </form> |
| **app.py**  @app.route('/thankyou') def thankyou():  user\_name = request.args.get('user\_name')  return render\_template('thankyou.html'**, full\_name=full\_name, validation\_errors=validate\_user(user\_name))**  def validate\_user(user\_name):  validation\_errors =[]  if not user\_name[-1].isdigit():  validation\_errors.append("Username should end with number")  if not any(char.isupper() for char in user\_name):  validation\_errors.append("Username should a contain uppercase letters")  if not any(char.islower() for char in user\_name):  validation\_errors.append("Username should a contain lowercase letters")  return validation\_errors |
| **thankyou.html**  {%extends 'base.html' %}  {% block title %}Thank You!{% endblock %}  {% block content %}  <div class="row">  <div class="card">  <div class="card-body">  <div class="alert alert-danger" role="alert">  <ul>  **{%for err in validation\_errors%}**  **<li>{{err}}</li>**  **{%endfor%}**  </ul>  </div>  <p>Thank you for submitting the form. We will get back to you shortly.</p>  </div>  </div>  </div>  {% endblock %} |

## FORMS IN FLASK

* **Step 1**: Flask provides a **FlaskForm** class that allows us to define forms and fields using Python classes.

|  |  |
| --- | --- |
| Install the Flask-WTF extension to use Flask forms | **pip install Flask-WTF** |

* **Step 2**: We first need to configure a secret key for security. This is required for CSRF protection provided by Flask-WTF
* **Step 3:** **Create a WTF form class**,
  + Create fields for each part of the form that we want to display on the template page.
* **Step 4: Set up a view function (.py)**
  + Add methods =[‘GET’,’POST’]
  + Create an instance of the form class
  + Handle form submission.

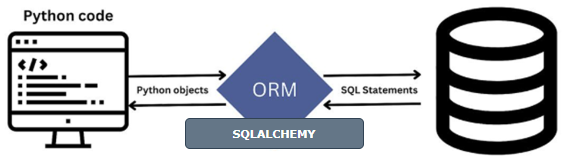
A diagram of a application

Description automatically generated

|  |  |
| --- | --- |
| **InfoForm.py**  **from flask\_wtf import FlaskForm** from wtforms import StringField, SubmitField  class **InfoForm**(**FlaskForm**):  breed = **StringField**("What Breed are you”)  submit = **SubmitField**("Submit") | |
| **app.py**  @app.route('/register', methods=['GET', 'POST']) def register():  breed = False  form = InfoForm()  if form.validate\_on\_submit():  breed = form.breed.data  print("breed", breed)  form.breed.data = ''  return render\_template(**'register.html', form=form, breed=breed**) |  |
| **register.html**  <div class="row">  <div class="card">  <div class="card-body">  {% if breed %}  <p>The Breed is {{breed}}</p>  {%else%}  <p>Please enter your breed</p>  {% endif %}  </div>  </div> </div> <form method="POST"> **{{form.hidden\_tag()}}  {{form.breed.label(class='form-label')}} {{form.breed(class='form-control')}}  {{form.submit(class='btn btn-primary')}}** </form> |

## DATABASE – SQLALCHEMY

### ORM



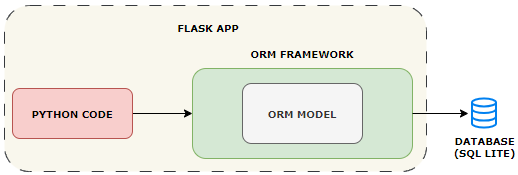
* To connect Python, Flask and SQL together we need ORM(Object Relation Mapper)
* An ORM will allow us directly use Python instead of SQL syntax to create, edit , delete and update the database
* The most common ORM for Python is **SQLAlchemy**

### SETTING UP THE ORM

* **Flask-SQLAlchemy** is an extension that allows for easy connection of Flask with SQLAlchemy

|  |  |
| --- | --- |
| **INSTALLL Flask-SQLAlchemy** | **pip install Flask-SQLAlchemy** |

### CRUD OPERATION

****

#### MODELS

* In an Object-Relational Mapping (ORM) framework, a model serves as an abstraction that represents a table in a database.

|  |
| --- |
| * Models directly link to a table in SQL database * We don’t need to create the table manually using SQL query, instead we create the Model class in Python that generates the table for us. |

#### STEPS TO CREATE MODEL CLASS

1. **Create a Model Class**
2. **Inherit it from db.Model**
3. **Provide the table Name (Optional) otherwise it takes class name**
4. **Add in table columns as attributes**
5. **Add in methods for \_\_init\_\_and \_\_repr\_\_**

## REST API

|  |  |
| --- | --- |
| INSTALL PACKAGE | **pip install Flask-Restful** |
| from flask import Flask from flask\_restful import Api, Resource app = Flask(\_\_name\_\_) api = Api(app)  **class Home(Resource):  def get(self):  print("message reached")  return {"message": "Welcome to the Home Page"}**  **api.add\_resource(Home, '/homepage-api')**  if \_\_name\_\_ == "\_\_main\_\_":  app.run(debug=True) | **API endpoint using Flask-RESTful.**   * The Flask application and Flask-RESTful API are initialized. * A resource class named Home is defined, which is a subclass of Resource. * The Home class has a get() method that handles GET requests to the /homepage-api endpoint. * Inside the get() method, a message is printed and a JSON response is returned with a welcome message. * The Home resource is added to the API using the add\_resource() method, with the /homepage-api endpoint as the URL path. |
|  | |

### CRUD APPLICATION

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| from flask import Flask from flask\_restful import Api, Resource app = Flask(\_\_name\_\_) **api = Api(app)**  # Format of users list - [ {'name': 'Alex'},{'name':'John'} ]  **users = []**  class User(Resource):  def **get**(self, name):  print("Get user", name)  for user in users:  if name == user['name']:  return user  **return {name: None},404 🡨 Setting status to 404**   def **post**(self, name):  user = {'name': name}  users.append(user)  return user   def delete(self, name):  deleted\_user ={}  for i, user in enumerate(users):  if name == user['name']:  deleted\_user = users.pop(i)  print("Deleted user", deleted\_user)  break  return deleted\_user  class AllUser(Resource):  def get(self):  return {'users': users} api.add\_resource(User, '/users/<string:name>') api.add\_resource(AllUser, '/users')  if \_\_name\_\_ == "\_\_main\_\_":  app.run(port=4000) | |  |  | | --- | --- | | GET | http://localhost:4000/users/Amit | | POST | http://localhost:4000/users/Avi | | DELETE | http://localhost:4000/users/Avi | | GET ALL | http://localhost:4000/users | |

## DEPLOYMENT – HEROKU