# RESTful API Development with Flask

A Practical Guide to Developing Your Own Web API Server Using Flask

**Learning Objectives**

By the end of this session, you will be able to:

- Set up and structure a Flask API project
- Implement a simple API capable of CRUD operations using Flask
- Learn best practices for building APIs

**Technical requirements**

Before starting, ensure you have the following softwares/tools installed and set up:

- **Visual Studio Code (VSCode)** – The code editor used for this session.
- **Postman Extension** – A lightweight Postman alternative to test API requests. This can be installed from the VSCode Extensions Marketplace.
- **Python (Version 3.8+)**
- **pip (Python package manager)** – Comes with Python but ensures it's installed.

**Session Overview**

We will be developing a **User Management API**, a simple RESTful API that allows you to **create, read, update, and delete (CRUD)** user data. This hands-on session will guide you through setting up the project, structuring it properly, and implementing the API endpoints using Flask.

The system contains five API endpoints, the first one has already been implemented as an example. You will be completing and testing the others as an exercise.

| **Method** | **Endpoint** | **Description** |
|---|---|---|
| GET | /api/users | Retrieve a lift of all users **(implemented)** |
| POST | /api/user | Create an new a user |
| GET | /api/user/**<id>** | Retrieve a user by ID |
| PUT | /api/user/**<id>** | Update a user by ID |
| DELETE | /api/user/**<id>** | Delete a user by ID |

# Section 1: Setup & installation

The project files are located in the **/user_api** sub folder under practical two materials. Please open this folder in **a new window** on VSCode and complete the following steps to set up your development environment.

**Step 1: Create & Activate a Virtual Environment**

**1.** Create a virtual environment

```
python -m venv env
```

**2.** Activate the virtual environment

Windows:

```
env\Scripts\activate
```

Mac/Linux:

```
source env/bin/activate
```

**Step 2: Install Dependencies**

In Practical 1, we installed the `requests` library using the `pip` command by specifying its name. Similarly, we can install multiple dependencies at once by specifying them in a text file.

To install the application's dependencies listed in the `requirements.txt` file, run:

```
pip install -r requirements.txt
```

**Step 3: Run the Application**

```
python run.py
```

# Section 2: Project Structure

A well-structured Flask project ensures scalability, maintainability, and ease of collaboration. We will use the following structure for our API:

```
user_api/
├── app/
│   ├── __init__.py
│   ├── extensions.py
│   ├── models/
│   │   └── user.py
│   ├── user/
│   │   ├── __init__.py
│   │   └── routes.py
├── config.py
├── run.py
├── requirements.txt
├── .env
├── .flaskenv
└── database.db # SQLite DB file will be created when you run the project
```

**1) app/ – Main Application Directory**
The **app/** folder holds all essential **resources** for the API, including models, routes, and extensions. It modularizes the project, making it scalable.

- **__init__.py**
  - Initializes the Flask app, registers essential extensions and API routes.
- **extensions.py**
  - Holds global instances for third-party packages (extensions) that are used to enhance functionality.

**2) app/models/ – Database Models**

The **models/** folder contains all database-related schemas and models using SQLAlchemy. SQLAlchemy is a Python SQL toolkit and **Object Relational Mapper (ORM)** that allows developers to interact with relational databases using Python classes instead of raw SQL queries, simplifying database management.

For this project, we will be saving user details to a simple SQLite database.

For more on SQLAlchemy information, see the Flask SQLAlchemy documentation.

**3) app/user/ – User Resource**

This folder contains the User resource. In Flask, a resource is created using Flask Blueprints, a modular approach used to separate functionality. Any additional modules, tools, or utility scripts related to this resource can be placed within this folder.

- **routes.py**
  - Defines all **User API endpoints**

**4) Other files / scripts**

- **config.py – Global Configuration Settings**
  - This file is responsible for managing centralized application settings, such as database connections and secret keys.

- **run.py – Application Entry Point**
  - This file serves as the main entry point for the Flask application. It typically initializes the app, loads configurations, and starts the development or production server.
- **.env.example – Environment Variables (Sensitive Data)**
  - This file provides a template for the **.env** file used in the project, such as database credentials, API keys, and secret keys.
  - This template is meant to help developers understand what environment variables are required for the project.
  - Before running the project you should create an **.env** file with the required environment variables.
  - **Important:** You should not commit your .env file to any public repositories.

- **.flaskenv – Flask Environment Variable (Optional)**
  - The **.flaskenv** file is specific to Flask and is used to configure environment variables that Flask automatically loads when running the app with the Flask CLI (flask run).

# Section 3: Building & Testing API Endpoints

Now, let's start implementing the API! The focus will be on modifying the **routes.py** file.

**How to Approach This Section**

- The first endpoint (GET /api/users) has already been implemented as an example.
- Study the example carefully to understand how it retrieves data using SQLAlchemy ORM.
- Your task is to implement the remaining four endpoints by completing the TODO sections.
- Test each endpoint in Postman to ensure it works correctly.
- Once done, compare your solution with the implementation on the **/solution** folder

**Why Use an ORM (Object-Relational Mapper)?**

ORMs like SQLAlchemy allow us to interact with the database using Python objects instead of raw SQL queries. This makes code cleaner, more maintainable, and database-agnostic.

Instead of writing:

```
SELECT * FROM users WHERE id=1;
```

We can simply use:

```
User.query.filter(User.id == 1).first()
```

This abstraction is **the power of ORMs**.

If you're not sure what method to use for a specific query, please see the Flask SQLAlchemy documentation and try looking things up before asking for help.

**Task:** Complete the Missing Endpoints in the **routes.py**