(start in visual studio code home page, open folder in desktop, open to main.py but also show the tabs for the other files)

For this video, I’m going to be talking about flask. Flask is a lightweight, micro web framework written in Python. It’s used to build web applications. I will be guiding you through a project you can complete if you have visual studio code or some kind of text editor installed on your computer, along with having python and flask installed on your terminal, which I’ll go through later in this video. For this project, we’ll be building a simple notes application. The goal is to create an application where we can submit notes and after submitting, you can see your completed note. Along with that feature, we will also be able to delete any of the notes that we create with the click of a button.

Instead of writing the code during the video, to save time, I’ll have it already complete and explain because each file is dependent on other files. I’ll run a quick demo right now

(run the file and demonstrate)

(exit website and go back to VS code)

(hit control c to exit)

Let’s ensure that we have the correct packages installed. Now in your terminal, run this command:

\*pip install flask flask\_sqlalchemy

For me, it’s going to say that the requirement is already satisfied since I already have it installed.

Before I explain the code in our files, I want to emphasis the importance of structuring our files, meaning where we should be saving each file. (talk about where each is saved only)

First, let’s go over the main.py file. Now I’m going to go back and forth a lot between these different files to show you how each file is importing certain functions from each other.

from website import create\_app

this imports the create\_app function from a folder or module called website. If I go to the \_\_init\_\_.py file, here we see the create\_app function, which is the application factory function that creates, configures, and returns the app. The reason it says from website is because the \_\_init\_\_.py file is located in the website folder. Another reason why it is very important where these files are saved at.

app = create\_app()

again, create\_app is a factory function that builds and returns a Flask app object with routes, configurations, and blueprints already set up. This line calls the function and creates the app instance. We will discuss the details of that function later in this video.

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

this line checks if you're running this script directly and not importing it as a module somewhere else. app.run starts a development server. debug=True means: the server will auto-reload if you make changes. you'll see detailed messages if anything goes wrong.

Now let’s go to our \_\_init\_\_.py file. This file is where the Flask application is created and configured, sets up the database and security, registers routes and functionality using blueprints, and makes sure your app is modular, clean, and scalable.

from flask import Flask

this imports the Flask class which you'll use to create your web application

from flask\_sqlalchemy import SQLAlchemy

this imports the SQLAlchemy object which helps us interact with databases using Python classes instead of raw SQL

from os import path

this allows checking whether a file exists (used when creating the database file).

db = SQLAlchemy()

this initializes the db object globally so it can be used across your app

DB\_NAME = "notes.db"

This is the name of your SQLite database file

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

this creates an instance of the Flask web application. \_\_name\_\_ tells Flask where to find your code and static files. If you're running this script directly, \_\_name\_\_ will be "\_\_main\_\_".

app.config['SECRET\_KEY'] = 'devkey'

the secret key protects your session data and cookies from being tampered with. It’s used for securely signing session cookies and other security-related needs

app.config['SQLALCHEMY\_DATABASE\_URI'] = f'sqlite:///{DB\_NAME}'

this sets up the database URI. It tells SQLAlchemy to use SQLite and save the database file to notes.db

db.init\_app(app)

this links the db object to this specific app instance

from .routes import routes

when you see a . before a file, that means it’s pulling from the routes file that we created. And it’s importing routes from that file. I will explain when we get to the routes file.

app.register\_blueprint(routes)

a Blueprint is a way to organize your routes and views into different files. the blueprint system helps break up a large app into modular, reusable components

with app.app\_context():

from .models import Note

db.create\_all()

you import your data models here so SQLAlchemy knows about them when creating the database. The create\_all() method creates all tables defined in your models.

Now let’s go to our routes.py file. This file is also named views.py. This file defines the routes (URLs) and view functions - basically, how your app responds to different webpage requests. In Flask, when someone visits a URL like /, Flask looks for a function decorated with @app.route('/')(hover over that decorator). These view functions return HTML pages, redirects, or JSON responses

from flask import Blueprint, render\_template, request, redirect

Blueprint allows you to separate different parts of your Flask app into reusable modules. In this case, it defines a set of routes. render\_template is used to render HTML files from a folder called templates. request gives access to form data submitted by the user. redirect sends the user to another route after an action

from .models import Note

this is a model that represents a row in the notes table. This will become more clear when we see the models.py file.

from . import db

imports the SQLAlchemy db object, which is your SQLAlchemy connection instance that was initialized in \_\_init\_\_.py

routes = Blueprint('routes', \_\_name\_\_)

this creates a Blueprint object named "routes". It groups a set of views (routes), which will be registered in your app through create\_app() in \_\_init\_\_.py

@routes.route('/', methods=['GET', 'POST'])

this is a route decorator. It tells Flask that this function should be triggered when the user visits the root URL (/) of the web app. This route handles both GET (when a user visits the page) and POST (when a form is submitted) requests

def home():

defines a view function called home. This is what gets called when a user accesses /.

if request.method == 'POST':

this is the if condition for when a user posts a new note

note\_content = request.form.get('note')

gets the value from a form input field named note from the HTML file

new\_note = Note(content=note\_content)

creates a new Note object derived from models.py

db.session.add(new\_note)

db.session.commit()

these two lines save the note to the database

return redirect('/')

after submitting, this statement sends the user back to the home page so the page refreshes and shows the new note

all\_notes = Note.query.all()

gets all notes from the database

return render\_template('home.html', notes=all\_notes)

renders an HTML page named home.html (must be inside the /templates folder) and passes it a variable called notes containing all the notes. This HTML file can also contain dynamic content using Jinja, the templating engine that Flask uses. I’ll go more in depth on this feature when we get to the html file.

@routes.route('/delete-note/<int:id>')

this route is triggered when the user goes to /delete-note/5 for example. <int:id> captures an integer from the URL and passes it to the function

note = Note.query.get(id)

retrieves the note with that ID

db.session.delete(note)

db.session.commit()

these two lines deletes the note from the database

return redirect('/')

takes the user back to the home page

Now let’s go to models.py. This is where you define the data models for your Flask app using SQLAlchemy. Models represent the tables in your database. SQLAlchemy is the most commonly used Object Relational Mapper (ORM) in python. It lets you interact with a database (like SQLite, PostgreSQL, or MySQL) using python classes instead of raw SQL.

from . import db

again, this line was also used in routes.py which imports the SQLAlchemy db object, which is your SQLAlchemy connection instance that was initialized in \_\_init\_\_.py

class Note(db.Model):

this defines a new table called Note. db.Model tells SQLAlchemy that this class maps to a table

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

this is primary key for each note. (auto-incremented by default). defines a column id as an integer

content = db.Column(db.String(1000))

defines a column content to store the text of the note. It’s limited to 1000 characters

Now let’s go to home.html. This file is an HTML template used in a Flask application to display and manage notes. This file uses Jinja2 templating for dynamic content and Bootstrap 5 for styling. Jinja allows us to use python code in HTML.

<!DOCTYPE html>

declares the document type. this tells the browser it's an HTML5 document

<html lang="en">

opens the HTML tag and sets the document language to English

<title>Flask Notes</title>

sets the title of the browser tab

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css">

links to the Bootstrap 5 CDN for prebuilt CSS styling. This gives your page modern, responsive styles without having to write custom CSS

<body class="p-5">

adds padding (p-5) around the whole body for spacing

<div class="container">

centers the content with appropriate margins and width, using Bootstrap's container class

<form method="POST">

sends the data via POST to the same route (/) when submitted

<textarea name="note" class="form-control mb-2" placeholder="Write a note..." required></textarea>

a multi-line text input. The name="note" attribute is important because it tells Flask what to look for in request.form.get("note") (go back to routes.py to show that line)

<button type="submit" class="btn btn-primary">Add Note</button>

submits the form. Styled using Bootstrap classes

<ul class="list-group mt-4">

creates a styled list with Bootstrap spacing (mt-4 = margin-top 4)

{% for note in notes %}

this starts a loop over the notes list passed from Flask to the template. Notice how this is python code in an HTML document. In this case we have to surround python code with curly brackets for it to go into effect. (go to routes.py to show the render\_template)

<li class="list-group-item d-flex justify-content-between align-items-center">

flexbox layout for spacing content and aligning vertically

{{ note.content }}

Jinja2 expression to insert the content of each note

<a href="/delete-note/{{ note.id }}" class="btn btn-sm btn-danger">Delete</a>

{{ note.id }}: used to build a link that points to a delete route like /delete-note/1. (go to routes to show delete-note/<int:id>). btn-danger: red delete button

So as you can see, the files all depend on one another and we’re able to use python code to develop a web application.