Rajesh Patil
GitHub: https://github.com/rajeshpatil88

FLASK API:

Flask is a lightweight and powerful web framework in Python that allows you to build APIs (Application Programming Interfaces) quickly and easily. It provides the necessary tools and functionalities to create web services that can handle HTTP requests and responses.

Basic commands to create a Flask API:

1. Install Flask: Before getting started, ensure you have Python installed. Then, open your terminal or command prompt and install Flask using pip:
pip install Flask

2. Create a Simple Flask App: To create a basic Flask app, create a new Python file (e.g., app.py) and add the following code:
from flask import Flask, render_template

```
app = Flask(__name__)

@app.route('/')
  def hello():
     return "Hello, World!"

@app.route('/page')
  def index():
     var1 = "value1"
     var2 = "value2"
     return render_template('template_file.html', var1=var1, var2=var2)

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

This code creates a Flask app, defines a route ("/page") that renders an HTML template (template_file.html) using the render_template function. The template can access variables var1 and var2.

3. Run the Flask App: Save the app.py file and run the Flask app by executing the following command in your terminal:

```
python app.py
```

You will see an output similar to:

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

The Flask app is now running, and you can access it by visiting http://127.0.0.1:5000/ in your web browser. Visiting http://127.0.0.1:5000/api/user/johndoe will return JSON data for the user "johndoe".

4. HTTP Methods for a Machine Learning App:

In machine learning Flask app, you'll use different HTTP methods to perform specific tasks. Here's how each method is used:

a. GET Method - Retrieving Data:

The GET method is like asking for information. In your app, you'll use it to retrieve data from the server. For instance, when a user accesses a certain route, the app responds with data from your machine learning model. It's like a user asking, "What's the prediction for this input data?" Your app provides the answer.

```
@app.route('/api/predict/<input_data>', methods=['GET'])
def get_prediction(input_data):
    # Use your machine learning model to make a prediction
    prediction = model.predict(input_data)
    return jsonify({'prediction': prediction})
```

b. POST Method - Sending Data for Prediction:

The POST method is about sending data to the server. In your app, you'll use it to send data that needs prediction. Think of it as submitting a request to predict new data. Your app processes the data using the machine learning model and sends back the prediction result.

```
@app.route('/api/predict', methods=['POST'])
def predict_data():
    data = request.get_json()  # Get input data from the request body
    prediction = model.predict(data)
    return jsonify({'prediction': prediction})
```

c. PUT Method - Updating the Model:

The PUT method is used to update a resource on the server. In machine learning app, this could mean updating your model with new data or retraining it. You might use it to fine-tune your model's performance over time.

```
@app.route('/api/update-model', methods=['PUT'])
def update_model():
    new_data = request.get_json()  # Get new training data from the request body
    model.train(new_data)  # Retrain the model with new data
    return "Model updated successfully."
```

d. DELETE Method - Removing Old Data:

The DELETE method removes a resource from the server. In a machine learning context, it could be used to clear out old or unnecessary data. For example, you might want to delete outdated training examples to keep your model up to date.

```
@app.route('/api/delete-data/<data_id>', methods=['DELETE'])
def delete_data(data_id):
    # Delete the specified data from your dataset
    return "Data deleted successfully."
```

e. PATCH Method - Fine-tuning Predictions:

The PATCH method is like making small adjustments. In ml app, you could use it to fine-tune predictions based on user feedback. For example, if a user provides correctional feedback on a prediction, you can use PATCH to update your model slightly for improved accuracy.

```
@app.route('/api/adjust-prediction/<input_data>', methods=['PATCH'])
def adjust_prediction(input_data):
    correction = request.get_json()  # Get correction data from the request body
    updated_prediction = model.adjust_prediction(input_data, correction)
    return jsonify({'updated_prediction': updated_prediction})
```

These HTTP methods provide a structured way for your machine learning app to interact with the server, making predictions, updating models, and handling data effectively.

6. In Flask, the app = Flask(__name__) line and the if __name__ == '__main__': app.run(debug=True) condition serve important roles in creating and running your Flask application. Let's break down their purposes:

- app = Flask(__name__):

When you create a Flask application, you instantiate a Flask object. The Flask class is used to create this object, and it represents your web application. The __name__ parameter is a special variable in Python that refers to the name of the current module. In this case, it allows Flask to determine the root path of the application.

When you set app = Flask(__name__), you are creating an instance of the Flask class and naming it app. This instance becomes the core of your web application, and it is used to define routes, handle requests, and manage various aspects of your app's behavior.

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

The if __name__ == '__main__': condition is a common Python programming pattern used to ensure that a block of code is only executed when the script is run directly, not when it is imported as a

module into another script.

In the context of a Flask application, this pattern is used to start the development web server when you run your script directly. When you execute your Flask script (python app.py), Python assigns the special name __main__ to the script being executed. The if __name__ == '__main__': condition checks whether the script is being run directly and not imported as a module. If this condition is true, the app.run(debug=True) method is executed, which starts the Flask development server.

The debug=True argument in app.run() enables debug mode, which provides additional information and automatic reloading of the server whenever you make changes to your code. Debug mode is very useful during development but should not be enabled in production.

7. Deployment File Structure: Organize your Flask project into a structure shown below:

```
my_flask_app/
│
├── app.py
├── requirements.txt
│
├── my_app/
│   ├── __init__.py
│   ├── routes.py
│   ├── models/
│   │   ├── __init__.py
│   │   ├── ml_model.py
│   │   └── preprocessing.py
│   ├── templates/
│   │   └── index.html
│   └── static/
│       ├── css/
│       └── js/
│
└── data/
    └── model.pkl
```

my_app/ Directory: I recommend placing your Flask application code within its own directory (in this case, my_app/). This helps keep your project organized, especially as it grows.

- __init__.py: Initializes the Flask app.
- routes.py: Defines your app's routes and views.
- models/: Contains your machine learning model implementation and related code.
- templates/: Stores your HTML templates.
- static/: Contains static files like CSS and JavaScript for your web pages.

data/ Directory: This directory is suitable for storing data files, such as your trained machine learning model (model.pkl). It's a good practice to keep your data separate from your application code.