**Introduction to FastAPI**

FastAPI is a modern, fast (high-performance) web framework for building APIs with Python. It is based on standard Python type hints and uses **Pydantic** for data validation and serialization and **Starlette** for the web parts. It is known for its speed, developer-friendly syntax, and automatic interactive documentation generation using **Swagger UI** and **ReDoc**.

**Why Use FastAPI?**

1. **High Performance**: Comparable to Node.js and Go.
2. **Automatic Documentation**: Swagger UI and ReDoc are generated automatically.
3. **Type Safety**: Utilizes Python type hints, improving code readability and reducing errors.
4. **Built-in Validation**: Data validation and parsing using Pydantic.
5. **Asynchronous Support**: Built-in support for asynchronous programming (async/await).
6. **Easy to Use**: Minimal boilerplate and concise code.

**How is FastAPI different from other frameworks?**

FastAPI differs from other Frameworks due to its built-in API documentation, highly permanent nature, automatic generation of open API, support for asynchronous programming and JSON schema documentation.

What are the Top companies using FastAPI?

* Netflix
* Google
* Microsoft

**How do you define a route in FastAPI?**

* To define a route in FastAPI, use the following syntax @app.route() decorator followed by the HTTP method.
* Example:
* @app.route('/student/{student\_id}')
* defines GET route with a parameter student\_id

**Core Components of FastAPI**

* **Routes**: Routes define the API endpoints. You create them by using decorators like @app.get(), @app.post(), @app.put(), etc.

A screen shot of a computer code

Description automatically generated

**Request and Response Models**: FastAPI uses **Pydantic models** for validating incoming request data and serializing outgoing responses.

A screenshot of a computer program

Description automatically generated

**Path Parameters, Query Parameters, and Body Data**: You can define various types of parameters, including path parameters, query parameters, and request bodies.

A close-up of a code

Description automatically generated

**Dependency Injection**: FastAPI allows you to use a clean dependency injection system to manage resources like databases, caches, etc.

A screen shot of a computer code

Description automatically generated

**Background Tasks**: You can perform background tasks asynchronously using FastAPI’s built-in support for background tasks.

A computer screen shot of a program

Description automatically generated

**Automatic Interactive API Documentation**

FastAPI automatically generates documentation for your API. You get two interactive interfaces:

* **Swagger UI**: Available at /docs, allowing you to test your API directly from the browser.
* **ReDoc**: Available at /redoc, providing a more detailed and static documentation view.

**Type Annotations and Data Validation**

FastAPI uses **Pydantic** for data validation. You can define request and response models as Python classes with type annotations.

* **Request Body Validation**:

A screen shot of a computer code

Description automatically generated

**Query Parameters Validation**: FastAPI will automatically convert query parameters to the required type based on your function signature.

A close-up of a code

Description automatically generated

**Response Models**: FastAPI automatically serializes your return data into JSON, and you can specify a response model for validation as well.

A computer screen shot of a computer code

Description automatically generated

**Asynchronous Programming**

FastAPI supports **async** and **await** for building highly concurrent applications. You can use async def for async request handlers, making it suitable for I/O-bound tasks like database queries or HTTP requests.

A computer code with green and black text

Description automatically generated

**Security and Authentication**

FastAPI provides utilities for handling common security mechanisms like:

* OAuth2 and Password-based authentication
* JWT (JSON Web Tokens) authentication
* Dependency injection for security checks (like OAuth2 scopes)

Example of basic OAuth2 password flow:

A screenshot of a computer code

Description automatically generated

**Testing FastAPI Applications**

FastAPI works well with Python’s pytest for testing your application. You can use **TestClient** from **Starlette** (which FastAPI is built on) for making HTTP requests in tests.

A computer screen shot of a code

Description automatically generated

FastAPI can be deployed with any WSGI/ASGI-compatible server. However, since it’s an ASGI app, you typically deploy it using **Uvicorn** or **Hypercorn**.

A close-up of a text

Description automatically generated

**Performance**

FastAPI is one of the fastest web frameworks available for Python, even outperforming frameworks like Flask and Django in benchmarks. It achieves this speed due to its use of **Starlette** for the web server (which is built on top of ASGI) and **Pydantic** for model validation.

**When to Use FastAPI?**

FastAPI is ideal for:

* Building RESTful APIs
* Microservices architecture
* Machine learning APIs
* High-performance applications with asynchronous processing
* Projects that require automatic API documentation (Swagger UI, ReDoc)

**Running the Code:**

1. **Train the Model:**
   * First, run the model training script to generate the model file:

python MLModel.py

**Run the FastAPI App:**

* Then, run the FastAPI app:

uvicorn main:app --reload

Webpage with code

<https://www.kdnuggets.com/using-fastapi-for-building-ml-powered-web-apps>

**1. Integrating Jinja2Templates in Your Web Application**

Jinja2 is a templating engine for Python web applications, commonly used with frameworks like **FastAPI** and **Flask**. It allows you to create HTML templates where dynamic content (such as prediction results) can be injected into the page.

To use **Jinja2** templates in FastAPI, you need to:

* Install **Jinja2** (if it's not already installed):

pip install jinja2

* Integrate it into your FastAPI app by importing Jinja2Templates and specifying the directory that contains your HTML files.

**2. Set Up Jinja2Templates in FastAPI**

First, you'll need to create a templates directory where your HTML files (e.g., index.html, result.html) will reside.

Here’s a basic setup in **FastAPI** to use Jinja2 templates:

A screenshot of a computer program

Description automatically generated

from fastapi import FastAPI, Form, Request

from fastapi.templating import Jinja2Templates

from fastapi.responses import HTMLResponse

from pathlib import Path

app = FastAPI()

# Initialize Jinja2Templates and specify the templates directory

templates = Jinja2Templates(directory=str(Path(\_\_file\_\_).parent / "templates"))

In this code:

* Jinja2Templates(directory=...) initializes the Jinja2 template engine and points it to the folder where your HTML templates are stored.
* We use Path(\_\_file\_\_).parent to refer to the current file's directory and create the correct path for the templates directory.
* **3. Define Asynchronous Route for the Root URL (/)**
* Now, let’s create a route that will serve the index.html file at the root URL /. This is the landing page of your web app.

@app.get("/", response\_class=HTMLResponse)

async def home(request: Request):

return templates.TemplateResponse("index.html", {"request": request})

Here:

* The home function is an asynchronous route that listens to GET requests on /.
* It renders the index.html template and passes a request object, which is required by Jinja2 templates to function properly.
* The response\_class=HTMLResponse tells FastAPI that the response will be HTML.

In index.html, you can create a form for the user to input the iris flower measurements.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

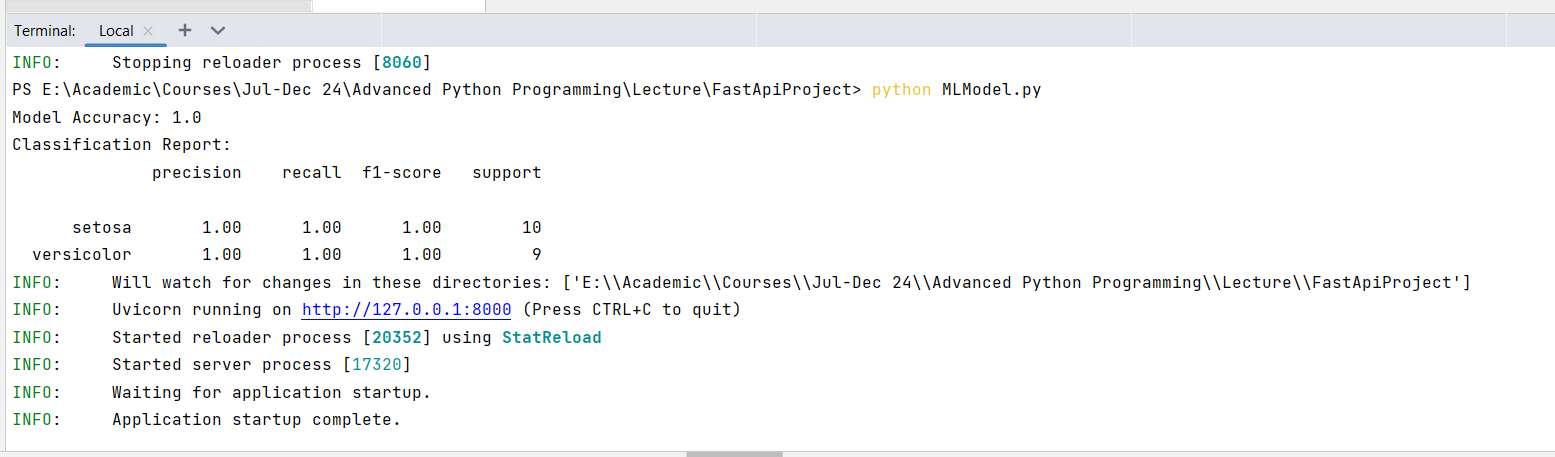
Description automatically generated

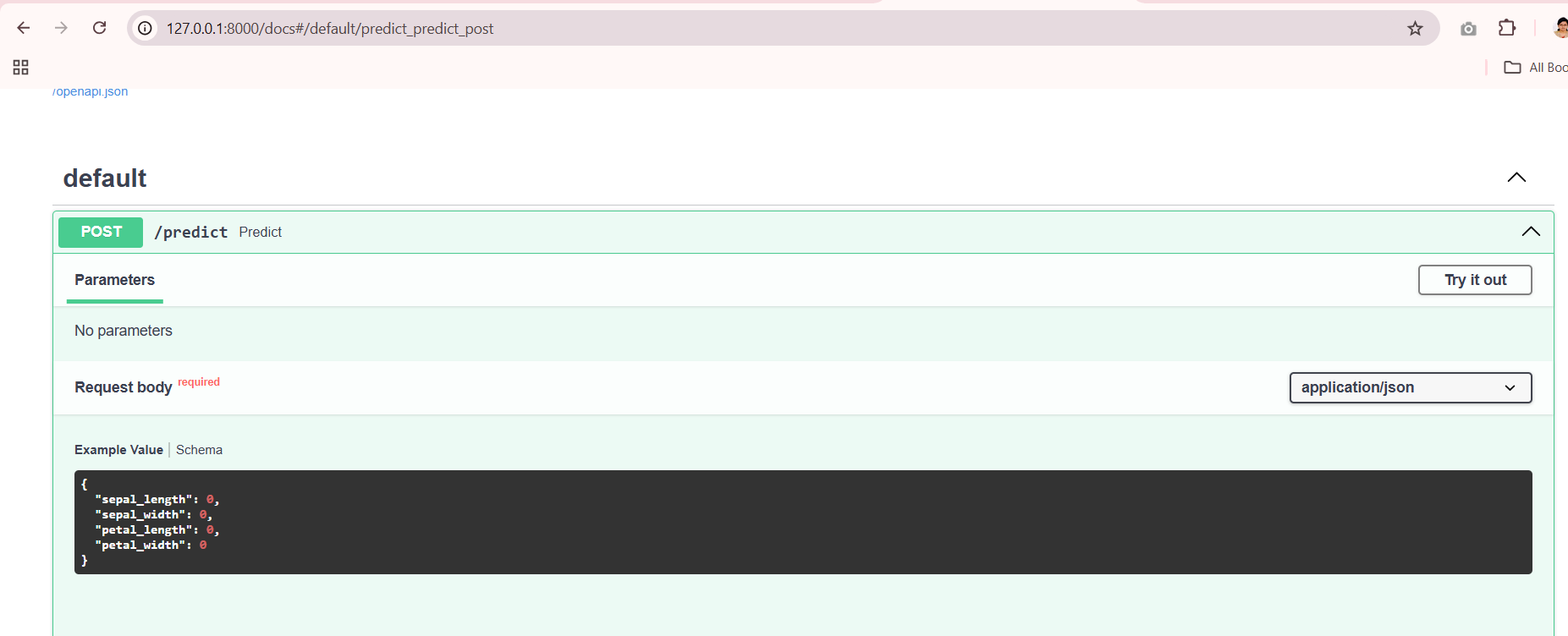
**MLModel.py**

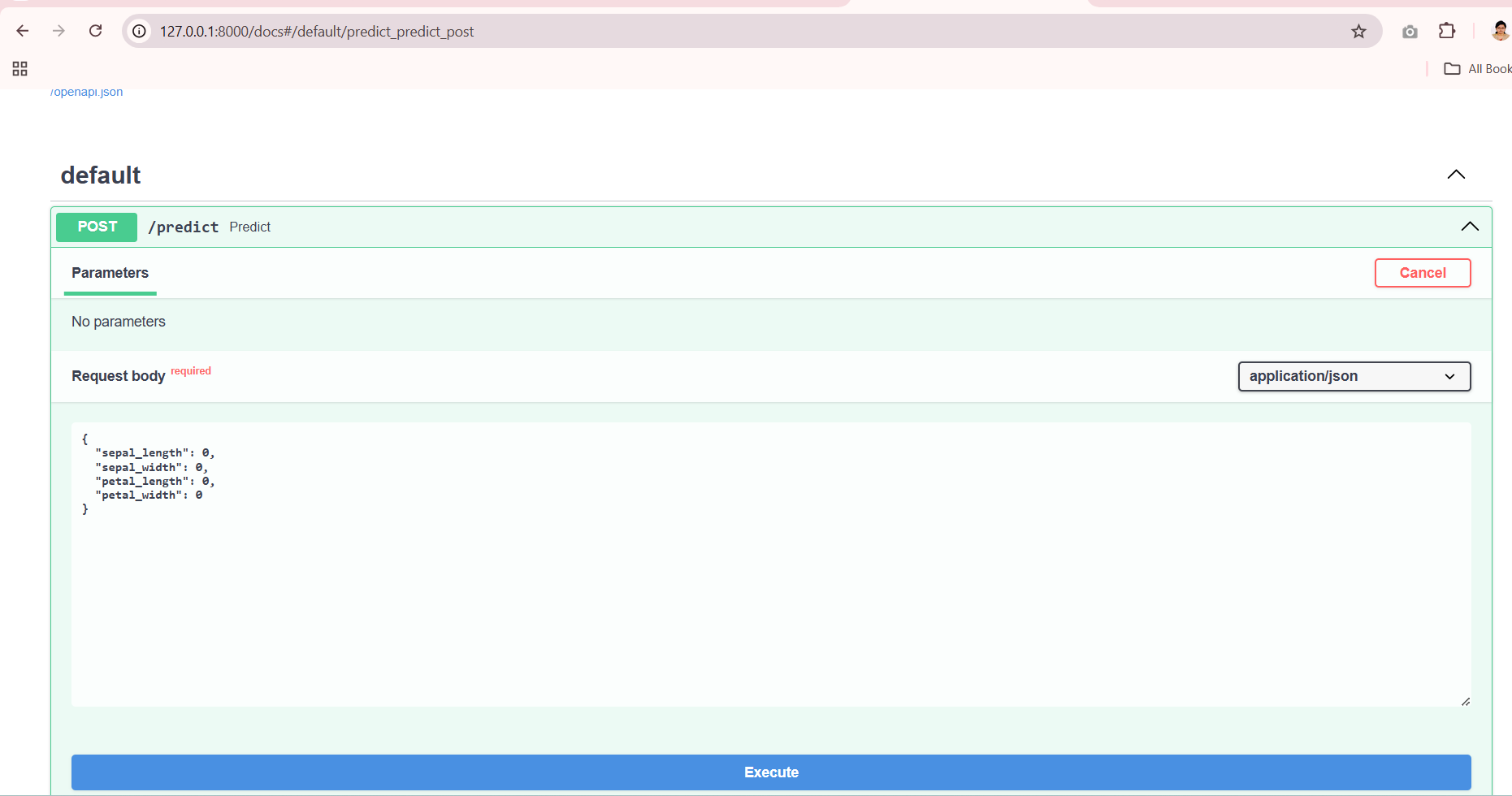
**import** numpy **as** np  
**from** sklearn.model\_selection **import** train\_test\_split  
**from** sklearn.ensemble **import** RandomForestClassifier  
**from** sklearn.metrics **import** accuracy\_score  
**import** joblib  
  
*# Step 1: Simple Dataset with 2 features  
# Features: [sepal\_length, sepal\_width]  
# Classes: [0 = Setosa, 1 = Versicolor]*X = np.array([  
 [5.1, 3.5], *# Setosa* [4.9, 3.0], *# Setosa* [6.2, 3.4], *# Versicolor* [5.9, 3.0], *# Versicolor* [5.8, 2.7], *# Versicolor* [5.1, 3.8] *# Setosa*])  
  
y = np.array([0, 0, 1, 1, 1, 0]) *# Labels for Setosa and Versicolor  
  
# Step 2: Split Data into Train and Test Sets*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33, random\_state=42)  
  
*# Step 3: Train Random Forest Model*model = RandomForestClassifier(n\_estimators=10, random\_state=42)  
model.fit(X\_train, y\_train)  
  
*# Step 4: Evaluate Model*y\_pred = model.predict(X\_test)  
accuracy = accuracy\_score(y\_test, y\_pred)  
print(**f"Model Accuracy: {**accuracy**:.2f}"**)  
  
*# Step 5: Save the Trained Model*joblib.dump(model, **"random\_forest\_model\_simple.pkl"**)

**Main.py**

**from** fastapi **import** FastAPI  
**from** pydantic **import** BaseModel  
**from** numpy **import** array  
**import** joblib  
  
*# Step 1: Create FastAPI Instance*app = FastAPI()  
  
*# Step 2: Load the Trained Model*model = joblib.load(**"random\_forest\_model\_simple.pkl"**)  
  
  
*# Step 3: Define Input Data Schema***class** InputData(BaseModel):  
 sepal\_length: float *# Feature 1: Sepal Length* sepal\_width: float *# Feature 2: Sepal Width  
  
  
# Step 4: Define Prediction Endpoint*@app.post(**"/predict"**)  
**def** predict(data: InputData):  
 *# Convert input features into numpy array (reshape to a 2D array for prediction)* input\_features = array([data.sepal\_length, data.sepal\_width]).reshape(1, -1)  
  
 *# Make prediction using the loaded model* prediction = model.predict(input\_features)  
 predicted\_class = int(prediction[0])  
  
 *# Map the class label to its corresponding name (for clarity)* class\_names = {0: **"Setosa"**, 1: **"Versicolor"**}  
 predicted\_class\_name = class\_names[predicted\_class]  
  
 **return** {**"predicted\_class"**: predicted\_class, **"class\_name"**: predicted\_class\_name}  
  
  
*# Optional: Health Check Endpoint*@app.get(**"/"**)  
**def** health\_check():  
 **return** {**"status"**: **"API is running"**}







A screenshot of a computer

Description automatically generated



**MLModel.py**

**from** sklearn.datasets **import** load\_iris  
**from** sklearn.model\_selection **import** train\_test\_split  
**from** sklearn.ensemble **import** RandomForestClassifier  
**from** sklearn.metrics **import** accuracy\_score, classification\_report  
**import** joblib  
  
*# Load the iris dataset*iris = load\_iris()  
X, y = iris.data, iris.target  
  
*# Split the data into training and testing sets*X\_train, X\_test, y\_train, y\_test = train\_test\_split(  
 X, y, test\_size=0.2, random\_state=42  
)  
  
*# Train a RandomForest classifier*clf = RandomForestClassifier(n\_estimators=100, random\_state=42)  
clf.fit(X\_train, y\_train)  
  
*# Evaluate the model*y\_pred = clf.predict(X\_test)  
accuracy = accuracy\_score(y\_test, y\_pred)  
report = classification\_report(y\_test, y\_pred, target\_names=iris.target\_names)  
  
print(**f"Model Accuracy: {**accuracy**}"**)  
print(**"Classification Report:"**)  
print(report)  
  
*# Save the trained model to a file*joblib.dump(clf, **"iris\_model.pkl"**)

main.py

**app.py**

**from** fastapi **import** FastAPI  
**from** pydantic **import** BaseModel  
**import** joblib  
**import** numpy **as** np  
**from** sklearn.datasets **import** load\_iris  
  
*# Load the trained model*model = joblib.load(**"iris\_model.pkl"**)  
  
app = FastAPI()  
  
  
**class** IrisInput(BaseModel):  
 sepal\_length: float  
 sepal\_width: float  
 petal\_length: float  
 petal\_width: float  
  
  
**class** IrisPrediction(BaseModel):  
 predicted\_class: int  
 predicted\_class\_name: str  
  
  
@app.post(**"/predict"**, response\_model=IrisPrediction)  
**def** predict(data: IrisInput):  
 *# Convert the input data to a numpy array* input\_data = np.array(  
 [[data.sepal\_length, data.sepal\_width, data.petal\_length, data.petal\_width]]  
 )  
  
 *# Make a prediction* predicted\_class = model.predict(input\_data)[0]  
 predicted\_class\_name = load\_iris().target\_names[predicted\_class]  
  
 **return** IrisPrediction(  
 predicted\_class=predicted\_class, predicted\_class\_name=predicted\_class\_name  
 )  
  
*# Optional: Health Check Endpoint*@app.get(**"/"**)  
**def** health\_check():  
 **return** {**"status"**: **"API is running"**}