FASTAPI Notes 02-June-2025

Overview

FastAPI is a Python web framework that's frequently used for building APIs and backend services. Articles on FastAPI cover a wide range of topics, including its features, benefits, tutorials, use cases, and comparisons with other frameworks like Flask and Django.

Key aspects covered in FastAPI articles:

* **Fundamentals:**

Articles often introduce FastAPI's core concepts, including its high performance, asynchronous capabilities, and reliance on standard Python type hints for data validation.

* **Features:**

They delve into specific features like automatic documentation generation (using OpenAPI/Swagger), integration with databases (SQLAlchemy, SQLModel), and middleware management.

* **Use Cases:**

Articles showcase how FastAPI is used for various applications, such as machine learning microservices, RESTful APIs, real-time applications, and IoT APIs.

* **Tutorials and Guides:**

Many articles provide step-by-step tutorials and guides on building different types of APIs, including those with authentication, database interactions, and more.

* **Comparisons:**

Some articles compare FastAPI with other frameworks like Flask and Django, highlighting its speed, ease of use, and other advantages.

* **Deployment:**

Articles also address the deployment aspects of FastAPI applications, including serverless deployment options like AWS App Runner and containerization with [FastAPI lambda container](https://fastapi.tiangolo.com/external-links/).

Example articles and resources:

* **Official FastAPI Documentation:**

The official FastAPI documentation provides a comprehensive tutorial, user guide, and reference material, including information on deployment, databases, and more.

* **Medium Articles:**

Articles on Medium offer various perspectives on FastAPI, from beginners' guides to advanced topics, such as building machine learning microservices.

* **Blogs:**

Blogs from tech companies like Microsoft and NVIDIA often feature tutorials and case studies on using FastAPI for specific applications.

* **Tutorials:**

Tutorials like [FastAPI Tutorial in Visual Studio Code](https://fastapi.tiangolo.com/external-links/) and [a FastAPI application monitoring example](https://fastapi.tiangolo.com/external-links/) provide practical experience in setting up and using FastAPI.

In summary, articles on FastAPI cover a wide range of topics, from basic introductions and tutorials to advanced use cases and comparisons with other frameworks, making it a valuable resource for developers looking to learn and use this powerful Python web framework.

The concept of **statelessness** in REST (Representational State Transfer) refers to the requirement that **each client request to the server must contain all the information needed to understand and process the request**, without relying on any stored context on the server.

### In simpler terms:

* **The server does not store anything about the client's state between requests.**
* Every request from the client must be **self-contained**.
* This allows each request to be **independent** and **reproducible**.

### Why is statelessness important?

1. **Scalability**: Since no client state is stored on the server, any server can handle any request. This enables load balancing and horizontal scaling.
2. **Reliability**: If one server fails, another can pick up requests without losing client context.
3. **Simplicity**: The server's design is simpler because it doesn’t need to manage sessions or remember previous interactions.
4. **Caching**: Stateless interactions are easier to cache, improving performance.

FastAPI Documentation

* <https://www.tutorialspoint.com/fastapi/fastapi_request_body.htm>
* <https://fastapi.tiangolo.com/tutorial/bigger-applications/#apirouter> multiple files API router

Set Up

* <https://fastapi.tiangolo.com/tutorial/first-steps/>
* <https://dev.to/snlucas/starting-with-fastapi-creating-a-super-simple-crud-exercise-37a0?signin=true> example post set up
* <https://www.tutorialspoint.com/fastapi/fastapi_request_body.htm> basics, concepts
* <https://fastapi.tiangolo.com/tutorial/query-params-str-validations/#import-query-and-annotated> next step query parameters
* <https://www.datacamp.com/tutorial/introduction-fastapi-tutorial>

Coding Examples

* <https://fastapi.tiangolo.com/tutorial/bigger-applications/#apirouter>
* Simple CRUD operations with Python and MongoDB <https://www.mongodb.com/developer/code-examples/python/python-crud-mongodb/>
* <https://www.datacamp.com/tutorial/introduction-fastapi-tutorial> advanced examples
* MongoDB and FastAPI example 2 - <https://www.freecodecamp.org/news/fastapi-quickstart/>
* <https://github.com/mongodb-developer/pymongo-fastapi-crud/blob/main/.github/workflows/main.yml> workflows
* A
* <https://medium.com/@vishnudas956783/fastapi-exposed-everything-you-should-know-about-pythons-fastest-framework-c9686efa7a93> more coding examples

Use Cases

* <https://medium.com/analytics-vidhya/fastapi-the-modern-age-api-framework-for-pythonista-4b2cd1e6652>

Frameworks

* <https://falcon.readthedocs.io/en/stable/>

[Falcon](https://falconframework.org/) is a minimalist ASGI/WSGI framework for building mission-critical REST APIs and microservices, with a focus on reliability, correctness, and performance at scale.

* <https://www.neoteroi.dev/blacksheep/getting-started/>

BlackSheep is an [ASGI](https://asgi.readthedocs.io/en/latest/) web framework, so it requires an ASGI HTTP server like [uvicorn](http://www.uvicorn.org/), or [hypercorn](https://pgjones.gitlab.io/hypercorn/)

PyPy is an alternate to CPython standard compiler It is JIT and 4X faster

* <https://doc.pypy.org/en/stable/install.html>

Venture Capital

* <https://blacksheep.ventures/>

FastAPI is a popular Python framework for building APIs, and it's well-suited for creating APIs for Large Language Models (LLMs). Several articles discuss how to use FastAPI to build LLM-powered APIs, covering topics like connecting to LLM providers like OpenAI, integrating with libraries like LangChain, and creating user-friendly interfaces for interacting with LLMs.

Key Aspects of Using FastAPI for LLM APIs:

* **High Performance:**

FastAPI's asynchronous capabilities and efficient handling of requests make it ideal for handling the computational demands of LLMs, especially for applications with real-time data processing and concurrent requests.

* **Ease of Use:**

FastAPI's simple syntax and automatic validation/serialization of data reduce boilerplate, making it easy to build APIs quickly.

* **Documentation and Type Safety:**

FastAPI provides built-in documentation generation (using Swagger UI) and leverages Python type hints, which enhances developer experience and makes it easier to understand and maintain APIs.

* **Integration with LLM Libraries:**

Articles demonstrate how to integrate FastAPI with libraries like LangChain and Langserve, which streamline the process of building LLM applications.  <https://www.linkedin.com/pulse/leveraging-fastapi-large-language-models-llms-guide-ganesh-jagadeesan-ad5ic/>

* **Security and User Functionality:**

Some articles cover implementing user authentication and security measures, ensuring that only authorized users can access and manage LLM-powered API endpoints.

Examples of FastAPI-based LLM APIs:

* **Smart Email Summarizer:**

Demonstrates how to use FastAPI to create an API that summarizes emails using an LLM (e.g., OpenAI's GPT).

* **Language Translation API:**

Shows how to build an API that translates text using LangChain and Langserve.

* **Code Analysis Assistant:**

Provides a step-by-step guide to building a code analysis assistant that utilizes an LLM (e.g., from Ollama).

* **AI Task Planner:**

Demonstrates how to build a local AI task planner using ClientAI and Ollama.

* **Personalized Email Writing Assistant:**

Explains how to build an API that helps users write personalized emails using an LLM.

Resources:

* [Medium](https://medium.com/@igorbenav/creating-llm-powered-apis-with-fastapi-in-2024-aecb02e40b8f) - "Creating LLM Powered APIs with FastAPI in 2024":

A series of articles on Medium that guides you through building LLM-powered APIs with FastAPI.

* [KDnuggets](https://www.kdnuggets.com/using-fastapi-for-building-ml-powered-web-apps) - "Using FastAPI for Building ML-Powered Web Apps":

Discusses how FastAPI can be used to build web applications powered by machine learning models.

* [DataCamp](https://www.datacamp.com/tutorial/serving-an-llm-application-as-an-api-endpoint-using-fastapi-in-python) - "Serving an LLM application as an API endpoint using FastAPI":

A tutorial on DataCamp that explains how to serve an LLM application as an API endpoint using FastAPI.

* [LinkedIn](https://www.linkedin.com/pulse/leveraging-fastapi-large-language-models-llms-guide-ganesh-jagadeesan-ad5ic) - "Leveraging FastAPI with Large Language Models (LLMs)":

A LinkedIn article that highlights the benefits of using FastAPI with LLMs.

Links

* <https://www.linkedin.com/pulse/leveraging-fastapi-large-language-models-llms-guide-ganesh-jagadeesan-ad5ic/>

Combining FastAPI with Large Language Models (LLMs) like OpenAI's GPT series can enable the development of sophisticated and high-performance applications that leverage advanced natural language processing (NLP) capabilities. This guide will explore the integration of FastAPI with LLMs in detail, highlighting key features, benefits, and practical applications.

### What is FastAPI?

FastAPI is a modern Python web framework designed for building APIs quickly and efficiently. It is characterized by:

* **Performance**: FastAPI is one of the fastest frameworks available, thanks to its support for asynchronous programming.
* **Automatic Documentation**: It provides interactive API documentation via Swagger UI and ReDoc.
* **Type Safety**: Utilizes Python type hints for robust data validation and serialization with Pydantic.
* **Asynchronous Support**: Built to handle high concurrency through asynchronous request handling.
* **Dependency Injection**: Simplifies the management of dependencies like database connections and authentication.

### Key Benefits of Combining FastAPI with LLMs

1. **High Performance** FastAPI's asynchronous capabilities ensure efficient handling of multiple requests concurrently, which is crucial when dealing with LLMs that may introduce latency due to their computational demands. This performance optimization helps in building responsive and scalable applications.
2. **Automatic Documentation** FastAPI automatically generates comprehensive and interactive documentation for your API endpoints. This documentation is crucial when integrating with LLMs as it provides an easy way to test and understand the API's functionality and parameters.
3. **Type Safety** FastAPI’s use of Python type hints and Pydantic ensures that input data is validated and serialized correctly. This is particularly important when interacting with LLMs, as it helps in maintaining the integrity of data passed to and from the model.
4. **Scalability** FastAPI is designed to be scalable, allowing you to handle increased loads and traffic effectively. This is beneficial for applications that use LLMs, which may need to handle a large volume of requests and generate responses in real-time.
5. **Security** FastAPI provides built-in tools for handling security, such as OAuth2 and JWT tokens. When working with LLMs, security features can help manage access to the API and protect sensitive data.

### Practical Applications of FastAPI with LLMs

1. **Chatbots** Build interactive chatbots capable of engaging in natural language conversations. FastAPI can handle user interactions and forward queries to the LLM for generating responses.
2. **Content Generation** Create tools for generating articles, blog posts, summaries, or creative writing. FastAPI can manage requests and handle the interaction with the LLM to generate and return content.
3. **Customer Support** Implement automated customer support systems that can handle common queries and provide instant responses. FastAPI can facilitate the backend logic, while the LLM handles natural language understanding and response generation.
4. **Data Analysis** Utilize LLMs for extracting insights from large volumes of textual data. FastAPI can manage API requests that process and analyze text data using the LLM.
5. **Language Translation** Develop applications that translate text from one language to another. FastAPI can manage the translation requests and interact with LLMs that support multiple languages.
6. **Personalized Recommendations** Create systems that provide personalized recommendations based on user input. FastAPI can handle the request logic and interact with LLMs to generate relevant suggestions.

### Advanced Considerations

1. **Rate Limiting and Caching** Implement rate limiting to manage API usage and avoid excessive calls to the LLM, which can be costly. Caching frequently requested results can also improve performance and reduce costs.
2. **Error Handling and Logging** Implement comprehensive error handling and logging to manage issues effectively and ensure reliable API operation. This is crucial when dealing with external services like LLMs.
3. **Scaling and Deployment** Consider deploying your FastAPI application on cloud platforms such as AWS, Azure, or Google Cloud for scalability. Use containerization with Docker and orchestration tools like Kubernetes for managing deployment.
4. **Data Privacy and Compliance** Ensure that your application complies with data privacy regulations, especially when handling sensitive or personal data. Implement appropriate security measures to protect user information.

### Conclusion

Integrating FastAPI with Large Language Models provides a powerful solution for building high-performance, advanced applications that leverage state-of-the-art natural language processing. FastAPI’s features, such as automatic documentation, type safety, and asynchronous support, complement the capabilities of LLMs, enabling you to create robust, scalable, and efficient APIs. By following the guidelines outlined in this guide, you can effectively harness the power of LLMs for a wide range of applications, from chatbots and content generation to customer support and data analysis.

**ToC**

[1] from Chatgpt

Here's a **simple FastAPI tutorial** to get you started with building APIs quickly and efficiently.

**Prerequisites**

Make sure you have:

* Python 3.7+
* pip (Python package manager)

Install FastAPI and an ASGI server (like Uvicorn):

bash

CopyEdit

pip install fastapi uvicorn

**Project Structure**

css

CopyEdit

fastapi-tutorial/

│

├── main.py

└── requirements.txt

**Step 1: Create a Basic FastAPI App**

**main.py**

python

CopyEdit

from fastapi import FastAPI

app = FastAPI()

@app.get("/")

def read\_root():

return {"message": "Welcome to FastAPI!"}

@app.get("/items/{item\_id}")

def read\_item(item\_id: int, q: str = None):

return {"item\_id": item\_id, "query": q}

**Step 2: Run the API Server**

bash

CopyEdit

uvicorn main:app --reload

Visit:

* Swagger docs: <http://127.0.0.1:8000/docs>
* ReDoc: <http://127.0.0.1:8000/redoc>

**Step 3: Add POST Method (Data Input)**

python

CopyEdit

from pydantic import BaseModel

class Item(BaseModel):

name: str

price: float

is\_offer: bool = None

@app.post("/items/")

def create\_item(item: Item):

return {"item": item}

**Step 4: Path, Query, and Body Parameters**

FastAPI automatically handles:

* Type validation
* Data parsing
* Swagger documentation

Example with multiple parameters:

python

CopyEdit

@app.put("/items/{item\_id}")

def update\_item(item\_id: int, item: Item, q: str = None):

return {"item\_id": item\_id, "item": item, "query": q}

**Step 5: Dependency Injection**

python

CopyEdit

from fastapi import Depends

def common\_parameters(q: str = None):

return {"q": q}

@app.get("/search/")

def search(params: dict = Depends(common\_parameters)):

return params

**Step 6: Organize with Routers**

Create a folder routers/ and a file items.py:

python

CopyEdit

# routers/items.py

from fastapi import APIRouter

router = APIRouter()

@router.get("/items/")

def get\_items():

return ["item1", "item2"]

Then include in main.py:

python

CopyEdit

from routers import items

app.include\_router(items.router)

**Extra Features**

FastAPI also supports:

* OAuth2 and JWT Auth
* Background tasks
* Middleware
* WebSockets
* SQL DB integration (SQLAlchemy / Tortoise ORM)

**Next Steps**

Would you like a tutorial including:

* SQLite or PostgreSQL database connection?
* JWT authentication?
* Async database calls?
* Docker support?

**Write a unit test to validate a specific row value or column-level transformation in a DataFrame**

import pandas as pd  
import unittest  
from unittest.mock import patch  
  
def transform\_dataframe(df, column\_name, transformation\_function):  
 """

Applies a transformation function to a specific column of a DataFrame.

Args:

df: The input DataFrame.

column\_name: The name of the column to transform.

transformation\_function: A function that takes a Series as input and returns a Series.

Returns:

A new DataFrame with the transformed column.

"""  
 df\_copy = df.copy()  
 df\_copy[column\_name] = df\_copy[column\_name].apply(transformation\_function)  
 return df\_copy  
  
class DataFrameTransformationTest(unittest.TestCase):  
 def test\_column\_transformation(self):  
 *# Create a sample DataFrame*  
 data = {'original': [1, 2, 3, 4, 5], 'other\_column': ['A', 'B', 'C', 'D', 'E']}  
 df = pd.DataFrame(data)  
  
 *# Define the transformation function (e.g., square the values)*  
 def square\_values(x):  
 return x\*\*2  
  
 *# Apply the transformation*  
 transformed\_df = transform\_dataframe(df, 'original', square\_values)  
  
 *# Verify the transformation on a specific row (e.g., row 1)*  
 expected\_value\_row1 = 2\*\*2 *# Square of the original value in row 1*  
 self.assertEqual(transformed\_df['original'][1], expected\_value\_row1, "Row 1 transformation failed")  
  
 *# Verify the transformation on the entire column*  
 expected\_transformed\_column = [1, 4, 9, 16, 25]  
 self.assertTrue(transformed\_df['original'].tolist() == expected\_transformed\_column, "Column transformation failed")  
  
 *# Verify that the other column remains unchanged*  
 self.assertEqual(transformed\_df['other\_column'].tolist(), df['other\_column'].tolist(), "Other column was modified")  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()