## **What is FastAPI?**

FastAPI is a modern web framework that is relatively fast and used for building APIs with Python 3.7+ based on standard Python-type hints. FastAPI also assists us in automatically producing documentation for our web service so that other developers can quickly understand how to use it. This documentation simplifies testing web service to understand what data it requires and what it offers. FastAPI has many features like it offers significant speed for development and also reduces human errors in the code. It is easy to learn and is completely production-ready. FastAPI is fully compatible with well-known standards of APIs (i.e. [OpenAP](https://www.geeksforgeeks.org/what-is-open-api-in-unix/)I and [JSON schema](https://www.geeksforgeeks.org/json-schema/)).

**Features of FastAPI**

* **Automatic Documentation:**FastAPI generates interactive API documentation automatically using the OpenAPI standard. You can access this documentation by visiting a specific endpoint in your application, which makes it incredibly easy to understand and test your API without having to write extensive documentation manually.
* **Python Type Hints**: One of FastAPI's standout features is its use of Python-type hints. By annotating function parameters and return types with type hints, you not only improve code readability but also enable FastAPI to automatically validate incoming data and generate accurate API documentation. This feature makes your code less error-prone and more self-documenting.
* **Data Validation:** FastAPI uses Pydantic models for data validation. You can define your data models using Pydantic's schema and validation capabilities. This ensures incoming data is automatically validated, serialized, and deserialized, reducing the risk of handling invalid data in your application.
* **Asynchronous Support:** With the rise of asynchronous programming in Python, FastAPI fully embraces asynchronous operations. You can use Python's async and await keywords to write asynchronous endpoints, making it well-suited for handling I/O-bound tasks and improving the overall responsiveness of your application.
* **Dependency Injection:** FastAPI supports dependency injection, allowing you to declare dependencies for your endpoints. This helps in keeping your code modular, testable, and maintainable. You can seamlessly inject dependencies like database connections, authentication, and more into your routes.
* **Security Features:**FastAPI includes various security features out of the box, such as support for OAuth2, JWT (JSON Web Tokens), and automatic validation of request data to prevent common security vulnerabilities like SQL injection and cross-site scripting (XSS) attacks.

**Installation and Setup of FastAPI**

To get started with FastAPI, you need to install Python, if not then [install Python3](https://www.geeksforgeeks.org/download-and-install-python-3-latest-version/). Then, you need to install fast API using the following command

$python -m venv myenv

$myenv\Scripts\activte

$pip install fastapi uvicorn pyd

$uvicorn main:app –reload

**Create a Simple API**

Here, we are creating a simple web service that says "Hello" when you visit a specific web address. With FastAPI, you can do this in just a few lines of code, To run this code, you can save it in a Python file, here we are saving the file as main.py.

**from** **fastapi** **import** FastAPI

*# Create a FastAPI application*

app = FastAPI()

*# Define a route at the root web address ("/")*

@app.get("/")

**def** read\_root():

**return** {"message": "Hello, FastAPI!"}

Now, execute the following command in your terminal:

uvicorn main:app --reload

**integrating PostgreSQL with FastAPI** and setting up the **repository layer** with a Python equivalent to **JPA/Hibernate**.

**✅ What You'll Build**

* PostgreSQL DB connection
* ORM with **SQLAlchemy** (Python's closest alternative to Hibernate/JPA)
* EmployeeRepository using SQLAlchemy for CRUD
* Integration with your existing service/controller structure

**⚙️ Step 1: Install Required Dependencies**

pip install sqlalchemy psycopg2-binary alembic

* sqlalchemy: ORM (like Hibernate)
* psycopg2-binary: PostgreSQL driver
* alembic: For DB migrations (optional but helpful)

async def get\_all\_employees(self):

        return self.employeesDataUploader.get\_employees()

    async def get\_employee(self, employee\_id: int):

        try:

            result =  list(filter(lambda emp: emp.id==employee\_id ,self.employeesDataUploader.get\_employees()))

            if result:

                result = result[0]

            print(f"Employee with ID {employee\_id}: {result}")

        except Exception as e:

            print(f"Error fetching employee with ID {employee\_id}: {e}")

            result = Employee()

        return result

    async def create\_employee(self, employee\_data: Employee):

        employees = self.employeesDataUploader.get\_employees()

        print(f"Current employees: {employees}")

        employees.append(employee\_data)

        return {"status": "Employee created successfully"}

    async def update\_employee(self, employee\_id: int, employee\_data: Employee):

        for index, emp in enumerate(self.employeesDataUploader.get\_employees()):

            if emp.id == employee\_id:

                self.employeesDataUploader.get\_employees()[index] = employee\_data

                return {"status": "Employee updated successfully"}

        return {"status": "Employee not found"}

    async def delete\_employee(self, employee\_id: int):

        self.employeesDataUploader.set\_employees(list(filter(lambda emp: emp.id != employee\_id, self.employeesDataUploader.get\_employees())))

        return {"status": "Employee deleted successfully"}