# Backend Concepts

Imagine you’re building a simple website where users can register with their name and email.

### Step 1: Frontend Sends a Request

Your frontend (built with HTML, JavaScript, or React) might have a form:

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Register</title>  
 <!-- <style>  
 body {  
 font-family: sans-serif;  
 display: flex;  
 flex-direction: column;  
 align-items: center;  
 justify-content: center;  
 height: 100vh;  
 margin: 0;  
 }  
 form {  
 display: flex;  
 flex-direction: column;  
 gap: 10px;  
 border: 1px solid #ccc;  
 padding: 20px;  
 border-radius: 5px;  
 }  
 #message {  
 margin-top: 20px;  
 font-weight: bold;  
 }  
 </style> -->  
</head>  
<body>  
 <h1>User Registration</h1>  
 <form id="registerForm">  
 <label for="name">Name:</label>  
 <input type="text" name="name" id="name" required/>  
   
 <label for="email">Email:</label>  
 <input type="email" name="email" id="email" required/>  
   
 <button type="submit">Register</button>  
 </form>  
  
 <div id="message"></div>  
   
 <!-- This is where the javascript begins to send the data to the backend -->  
 <script>  
 const form = document.getElementById('registerForm');  
 const messageDiv = document.getElementById('message');  
  
 form.addEventListener('submit', async (event) => {  
 event.preventDefault();  
  
 const formData = new FormData(form);  
 const name = formData.get('name');  
 const email = formData.get('email');  
  
 try {  
 const response = await fetch('/register', {  
 method: 'POST',  
 headers: {  
 'Content-Type': 'application/json',  
 },  
 body: JSON.stringify({ name, email }),  
 });  
  
 if (response.ok) {  
 const result = await response.json();  
 messageDiv.textContent = result.message;  
 form.reset(); // Clear the form fields  
 } else {  
 const error = await response.json();  
 messageDiv.textContent = 'Error: ' + (error.detail || 'An unknown error occurred.');  
 }  
 } catch (error) {  
 messageDiv.textContent = 'An error occurred during registration.';  
 }  
 });  
 </script>  
</body>  
</html>

When the user submits the form, the browser sends the data to your backend API.

### Step 2: FastAPI Receives the Request

In FastAPI, you might have this route in main.py:

from fastapi import FastAPI  
from pydantic import BaseModel  
  
app = FastAPI()  
  
class User(BaseModel):  
 name: str  
 email: str  
  
@app.post("/register")  
def register\_user(user: User):  
 # In a real app, you'd save this to the database  
 return {"message": f"User {user.name} registered with email {user.email}"}

FastAPI automatically reads the form data or JSON, turns it into a User object, and returns a response.

### Step 3: Connect to a Database

You can save the user data in a real database (like SQLite or PostgreSQL):

import sqlite3  
  
@app.post("/register")  
def register\_user(user: User):  
 conn = sqlite3.connect("users.db")  
 cursor = conn.cursor()  
 cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", (user.name, user.email))  
 conn.commit()  
 conn.close()  
 return {"message": f"User {user.name} registered!"}

This is a very basic example — in real projects you’d use SQLAlchemy or another ORM, but this shows the full flow:

**Frontend ➜ FastAPI ➜ Database ➜ Response back to user**

That’s how FastAPI connects everything together to build a real-world web app. In order to test this out, be sure to run this line of code to install the necessary libraries:  
***pip install fastapi uvicorn python-multipart pydantic sqlite3***

## What is FastAPI?

**FastAPI** is a modern, high-performance web framework for building APIs with Python 3.7+ based on standard Python type hints. It is designed to be fast, easy to use, and highly efficient, making it a great choice for building backend services.

## Why Use FastAPI?

* **Fast**: One of the fastest Python frameworks, powered by Starlette and Pydantic.
* **Async Support**: Supports async and await for high-performance asynchronous code. In Python, async and await are used to write asynchronous functions that can perform tasks like reading files or making web requests without stopping the rest of the program. This means your server can handle multiple requests at once more efficiently, which is important for scalability. high-performance asynchronous code.
* **Data Validation**: Built-in validation using Pydantic models.

## Step-by-Step Setup

### 1. Install Python

Make sure you have Python 3.7 or higher installed.

* You can download it from: <https://www.python.org/downloads/>

To check your version:

python --version

### 2. Create a Project Folder

mkdir rfp-backend  
cd rfp-backend

This will be the main directory for your backend code.

### 3. Create a Virtual Environment

A virtual environment keeps your project dependencies isolated.

python -m venv venv  
source venv/bin/activate # On Windows: venv\Scripts\activate  
#git bash source venv/Scripts/activate

### 4. Install FastAPI and Dependencies

pip install fastapi uvicorn python-multipart pydantic python-docx PyMuPDF openai  
pip freeze > requirements.txt

This installs FastAPI and some useful libraries:

* uvicorn: Used to run your app
* python-multipart: For handling file uploads
* python-docx and PyMuPDF: For parsing DOCX and PDF files
* openai: For GPT integration

### 5. Create Folder Structure

mkdir -p app/api/routes app/core app/services app/db app/models tests  
touch app/main.py

This organizes your code cleanly for larger projects.

## FastAPI Key Concepts (With Code Examples)

### 1. **FastAPI Application Instance**

from fastapi import FastAPI  
  
app = FastAPI()

This is your main app object. You will attach all your API routes to this.

### 2. **Basic Route (Endpoint)**

@app.get("/hello")  
def say\_hello():  
 return {"message": "Hello, world!"}

This defines a GET endpoint /hello that returns JSON.

### 3. **Pydantic Models for Validation**

from pydantic import BaseModel  
  
class User(BaseModel):  
 name: str  
 age: int

Pydantic helps validate incoming request data. If someone sends invalid data, FastAPI will reject it automatically.

### 4. **Handling Parameters**

@app.get("/users/{user\_id}")  
def get\_user(user\_id: int, include\_details: bool = False):  
 return {"user\_id": user\_id, "details": include\_details}

* user\_id is a **path parameter** (part of the URL)
* include\_details is a **query parameter** (after the ? in the URL)

### 5. **POST Request with JSON Body**

@app.post("/users")  
def create\_user(user: User):  
 return {"name": user.name, "age": user.age}

Here, user is passed as a JSON body. FastAPI validates it using the User model.

### 6. **Uploading Files**

from fastapi import File, UploadFile  
  
@app.post("/upload")  
async def upload\_file(file: UploadFile = File(...)):  
 contents = await file.read()  
 return {"filename": file.filename}

This lets users upload files such as PDFs and Word docs.

### 7. **Modular Routing**

Helps organize routes in separate files.

# app/api/routes/ping.py  
from fastapi import APIRouter  
  
router = APIRouter()  
  
@router.get("/ping")  
def ping():  
 return {"message": "pong"}

Then in main.py:

from fastapi import FastAPI  
from app.api.routes import ping  
  
app = FastAPI()  
app.include\_router(ping.router, prefix="/api")

### 8. **Running the App Locally**

uvicorn app.main:app --reload

* --reload watches for code changes
* app.main:app = [folder].[file]:[FastAPI app object]

## Common FastAPI Folder Structure

rfp-bot-backend/  
├── app/ # Main application code  
│ ├── api/ # API-specific code  
│ │ └── routes/ # Contains route files like user.py, upload.py  
│ ├── core/ # App configuration, settings, logging  
│ ├── services/ # Business logic, helpers, reusable functions  
│ ├── db/ # Database connection, models (SQLAlchemy), seed scripts  
│ ├── models/ # Pydantic data models for request/response schemas  
│ └── main.py # Entry point where FastAPI app is created  
├── tests/ # Unit and integration tests  
├── requirements.txt # Lists Python dependencies  
└── Dockerfile # Instructions for containerizing your app

## Summary

FastAPI is beginner-friendly yet powerful. You:

1. Define an app
2. Add routes (GET, POST, PUT, DELETE, etc.) — these are called HTTP methods, and each one has a specific use:
   * **GET**: Retrieve data (e.g., get a list of users)
   * **POST**: Send new data to the server (e.g., create a new user)
   * **PUT**: Update existing data completely (e.g., replace a user record)
   * **PATCH**: Update part of the data (e.g., just update the user’s name)
   * **DELETE**: Remove data (e.g., delete a user by ID)
3. Use Pydantic for data validation
4. Run the app with uvicorn