Build, Test, Deploy, Earn

FastAPI

Full Stack Mastery

Rajesh Kumar
www.geekyshows.com

Table of Contents

What is Framework

A framework may include predefined classes and functions that can be used to process input, manage hardware devices, and interact with system software.
The purpose of the framework is to allow developers to focus on building a unique feature for their Projects rather than writing code from scratch.
FastAPI which we are studying is a Web Framework.

Why use Framework

Using framework makes development fast and reliable along with below advantages:

- Collection of tools
- No need to start from scratch
- Save Time
- Improve Productivity
- Clean Code
- Reusable Code
- Testing
- Debugging

What is Web Framework

A Web Framework (WF) or Web Application Framework (WAF) which helps to build Web Applications.
Web frameworks provide tools and libraries to simplify common web development operations. This can include web services, APIs, and other resources.
Web frameworks help with a variety of tasks, from templating and database access to session management and code reuse.
More than 80% of all web app frameworks rely on the Model View Controller architecture.
Few Web Framework Examples are: FastAPI, Laravel, Codeigniter, Zend, Django, Spring etc.

What is FastAPI

FastAPI is a modern, high-performance web framework for building APIs with Python 3.6+ that emphasizes speed, ease of use, and automatic documentation. It is particularly designed for creating RESTful APIs, utilizing Python's standard type hints and the Pydantic library for data validation, serialization, and deserialization.

Features and Advantages of FastAPI

- Automatic interactive API documentation
- Asynchronous support
- Built-in data validation
- Dependency injection system
- Developer-friendly
- Security

FastAPI Prerequisite

Knowledge You should have

- HTML
- CSS
- JavaScript
- SQL
- Python
- PIP

Software Requirements

- Python 3.10 or Higher
- PIP
- Text/Code Editor/IDE – Notepad++, VS Code, ATOM, Brackets, PyCharm
- Web Browser – Google Chrome, Mozilla Firefox, Edge

Does FastAPI follow MVT or MVC

No, FastAPI does not follow the MVT (Model-View-Template) architecture like Django does.
FastAPI is API-first, not template-focused.
FastAPI Pattern is Closer to Modular Clean Architecture or MVC-lite:

- Models → Pydantic models or SQLAlchemy ORM models
- Routers / Controllers → Handle routing and API Route logic
- Schemas → Request/response validation (Pydantic)
- Services → Business logic layer
- Templates (optional) → Can be added manually using Jinja2

What we can build with FastAPI

FastAPI is increasingly popular among high-profile organizations due to its performance, asynchronous features, and developer-friendly design. We can build web applications similar to:

- Netflix
- Microsoft
- Uber
- SendCloud
- OnlineDoctor

Check FastAPI is installed or Not

fastapi --version

How to Install FastAPI

- Create Virtual Environment (VE)
    - python -m venv venv

- Activate Virtual Environment (VE)
    - source venv/bin/activate - Linux or Mac
    - .\venv\Scripts\activate - Windows
- Install FastAPI
    - pip install "fastapi[standard]"

FastAPI Project

Setting up your first FastAPI project involves a combination of essential files, directories, and best practices to ensure maintainability, scalability, and development efficiency.

- app/main.py - Main FastAPI app and endpoints
- requirements.txt - Dependencies
- .env - Environment configuration
- app/product/ - All files related to product (optional)
- app/core/ - All Files related to basic Settings, configuration (optional)
- app/db/ - All files related to Database configurations

How to run development server

FastAPI provides built-in server which we can use to run our project.
fastapi dev – This command is used to run built-in server of FastAPI.
Steps: -

- Go to Project Folder
- Then run command ***fastapi dev app/main.py OR fastapi dev***

Server Started, Visit http://127.0.0.1:8000 or http://localhost:8000

You can specify Host and Port number

fastapi dev app/main.py --host 0.0.0.0 –port 5555

Visit http://0.0.0.0:5555

To Stop the Server Press **CTRL+C**

Internally fastapi dev uses uvicorn so You can also run below commands to run server

uvicorn app.main:app –reload

uvicorn app.main:app --reload --host 0.0.0.0 --port 5555

Path Parameters

When building web APIs, the most natural way to access dynamic data is through the URL. In FastAPI, this is achieved using path parameters, a core concept that allows us to capture values from the URL itself and use them in our logic.

What Are Path Parameters

Imagine an API endpoint like: */items/42*

Here, 42 is not a fixed path. it's a variable, also called a path parameter. FastAPI lets you define routes like this:

@app.get("/items/{item_id}")

async def read_item(item_id):

    return {"item_id": item_id}

When a request is made to /items/42, FastAPI captures the 42 and passes it to the read_item function as the item_id.

Declaring Path Parameters with Type Hints

FastAPI becomes even more powerful when you add type annotations to your path parameters:

@app.get("/items/{item_id}")

async def read_item(item_id: int):

    return {"item_id": item_id}

Now, FastAPI will:

Convert the value "42" from the URL to an integer.

Validate the type, returning an error if it's not a valid int.

Order of Routes Matters

FastAPI processes routes in the order they’re defined. Consider:

@app.get("/users/me")

def get_current_user(): ...

@app.get("/users/{user_id}")

def get_user(user_id: str): ...

If you reverse these, /users/me would be treated as user_id = "me". Always define more specific routes before generic ones.

Fixed Values with Enums

Sometimes, you want to limit a parameter to predefined values for example, model types like alexnet, resnet, or lenet.

FastAPI supports this with Python’s Enum:

class ModelName(str, Enum):

    alexnet = "alexnet"

    resnet = "resnet"

```
    lenet = "lenet"


@app.get("/models/{model_name}")

async def get_model(model_name: ModelName): ...
```

FastAPI will:

- Validate input against allowed values.
- Show options in the interactive docs.
- Auto-generate detailed OpenAPI specs.

Path Parameters That Contain Paths

You might want a route like: GET /files/home/user/data.txt

To handle slashes within a parameter, FastAPI supports a path converter:

```
@app.get("/files/{file_path:path}")

async def read_file(file_path: str):

    return {"file_path": file_path}
```

This allows file_path to include slashes and directories. In this case, /files/home/user/data.txt will pass "home/user/data.txt" to the function.

If you need a leading slash, use a double-slash in the URL: /files**//**home/user/data.txt

Query Parameters

APIs often need to filter, paginate, or search data dynamically. While path parameters represent fixed resource identifiers, query parameters give your clients a way to modify or fine-tune their requests. FastAPI makes handling query parameters both intuitive and powerful.

What Are Query Parameters

Query parameters are the part of the URL that comes after the ? symbol, used to pass optional key-value pairs. For example: http://127.0.0.1:8000/items/?skip=0&limit=10

Here, skip and limit are query parameters.

Declaring Query Parameters

In FastAPI, any function argument that is not a path parameter, Has a default value is treated as a query parameter automatically.

```
@app.get("/items/")

async def read_items(skip: int = 0, limit: int = 10):

    return fake_items_db[skip : skip + limit]
```

skip and limit are not part of the path. They're optional and have default values.

If the user doesn't provide them, FastAPI uses the defaults.

Type Conversion and Validation

FastAPI uses type hints (int, str, bool, etc.) to :

- Parse query strings into correct types
- Return automatic 422 errors if parsing fails

```
@app.get("/items/")
async def read_items(skip: int = 0):
```

Optional Query Parameters

Want a query parameter to be optional? Just assign None as the default value:

```
@app.get("/items/{item_id}")
async def read_item(item_id: str, q: str | None = None):
    if q:
        return {"item_id": item_id, "q": q}
    return {"item_id": item_id}
```

Calling /items/abc returns: {"item_id": "abc"}

Calling /items/abc?q=search returns: {"item_id": "abc", "q": "search"}

Boolean Query Parameters

FastAPI supports automatic parsing of booleans too:

```
@app.get("/items/{item_id}")
async def read_item(item_id: str, short: bool = False): ...
```

Accepts:

- short=true
- short=1
- short=on
- short=yes

…and many other variations (case-insensitive).

Multiple Path + Query Parameters

You can combine multiple types of parameters in one route:

```
@app.get("/users/{user_id}/items/{item_id}")
async def read_user_item(user_id: int, item_id: str, q: str | None = None, short: bool =
False):
```

- user_id and item_id → path parameters
- q and short → query parameters

FastAPI figures this out automatically based on:

- Route declaration
- Type hints
- Default values

Required Query Parameters

Any parameter without a default is required.

```
@app.get("/items/{item_id}")

async def read_item(item_id: str, needy: str):

    return {"item_id": item_id, "needy": needy}
```

Calling /items/abc (without needy) will not work, but calling /items/abc?needy=value works.

Response Status Codes

When building APIs, it's not just about what you return, it's also about how you return it. The HTTP status code tells the client the result of their request. FastAPI makes it easy to set these codes for each route.

Declaring a Response Status Code

FastAPI allows you to explicitly define the HTTP status code returned by an endpoint. This is done directly inside the path operation decorator (@app.get(), @app.post(), etc.).

```
from fastapi import FastAPI

app = FastAPI()

@app.post("/items/", status_code=201)

async def create_item(name: str):

    return {"name": name}
```

status_code is passed to the decorator, not the function itself.

It tells FastAPI (and the OpenAPI docs) what to expect from this operation.

Range	Meaning	Example Codes
100 – 199	Informational	101 Switching Protocols
200 – 299	Success	200 OK, 201 Created
300 – 399	Redirection	302 Found
400 – 499	Client Errors	400 Bad Request
500 – 599	Server Errors	500 Internal Server Error

Use Enums Instead of Raw Numbers

You don’t have to memorize status codes. FastAPI provides readable constants via fastapi.status (which comes from Starlette).

```
from fastapi import FastAPI, status

app = FastAPI()

@app.post("/items/", status_code=status.HTTP_201_CREATED)

async def create_item(name: str):

    return {"name": name}
```

This makes your code more readable, helps with autocomplete, and avoids mistakes.

Request Body and Pydantic Model

In the context of HTTP, a request body is the data a client (e.g., browser or mobile app) sends to the server when making a request. This is common in operations like creating or updating data.

Unlike path or query parameters, the request body is not visible in the URL and can contain structured data such as JSON, XML, etc. FastAPI supports request bodies primarily through Pydantic models.

Why Use Request Bodies

Many operations especially in RESTful APIs require a client to send structured data:

- POST to create a resource
- PUT or PATCH to update a resource

FastAPI makes handling these request bodies easy and robust by validating the structure and types automatically.

GET requests should not have a body, but FastAPI technically allows it (rare edge cases).

FastAPI's auto-generated docs (Swagger UI) will not display body fields for GET endpoints.

What is Pydantic

Pydantic is a popular Python library for data validation and settings management. It makes use of Python's type annotations to validate, parse, and serialize data structures. By defining models as Python classes, you can ensure that the data your application receives or processes adheres to the expected types and structure, catching inconsistencies early and providing meaningful error messages.

Pydantic’s Role in FastAPI

FastAPI is a modern Python web framework for building APIs, known for its speed and use of modern Python features. Pydantic is fundamental to FastAPI almost all request and response validation, parsing, and serialization is powered by Pydantic models.

How Pydantic Powers FastAPI

- Request Validation - When a client sends data (such as JSON in a POST request), FastAPI validates the incoming data against the Pydantic model provided in the endpoint definition. Only data matching the model's schema and types passes through; otherwise, FastAPI returns clear error messages.
- Automatic Data Parsing - FastAPI converts request data into Pydantic model instances, so you work with regular Python objects in your endpoint code making code more robust and type-safe.
- Response Serialization - You can specify a Pydantic model as your response schema using FastAPI's response_model parameter. FastAPI returns JSON responses that strictly match the Pydantic model, ensuring that clients always receive consistent data structures.
- Documentation Generation - Because FastAPI knows your API's data shapes (through Pydantic), it can auto-generate detailed and accurate OpenAPI documentation, which is visualized via Swagger UI, benefiting developers consuming your API.

Multiple Body Parameters

In the real world, most API endpoints require complex input sometimes a combination of structured data, query strings, and path variables. FastAPI makes this easy by allowing you to combine multiple types of parameters, including multiple body parameters, path, and query inputs.

Mixing Path, Query, and Body Parameters

FastAPI is smart enough to distinguish between path parameters, query parameters, and body parameters based on their type annotations, their position in the function signature, and whether they are Pydantic models or simple types.

- You can freely combine path, query, and body parameters.
- You can declare multiple body parameters using Pydantic models.
- Use Body() to explicitly treat simple types as body parameters.
- Use embed=True to nest a single Pydantic model inside a body key.

Body Field

In FastAPI, just like you can use Query, Path, and Body to add validation and metadata to function parameters, you can use Field from Pydantic to do the same inside your Pydantic models. This enhances Validation behavior, OpenAPI schema generation and Documentation clarity.

Cookie Parameters

In HTTP, cookies are small pieces of data stored on the client side and sent automatically with every request to the same domain. FastAPI provides built-in support to read and write cookies through its dependency system and response utilities.

Cookies are commonly used for:

- Session management (e.g. session ID)

- Personalization (e.g. user preferences)
- Tracking (e.g. analytics IDs)

Unlike query parameters or path variables, cookies are not part of the URL. They are automatically sent by the browser, often without developer intervention.

FastAPI allows you to:

- Read incoming cookies as function parameters
- Group cookie parameters into Pydantic models
- Set cookies in the response

Header Parameters

FastAPI supports the declaration of header parameters in a way that mirrors how you work with path, query, and cookie parameters. These headers are extracted from incoming HTTP requests and passed as function parameters in the route handlers.

- Use Header() to declare header parameters just like Query() or Path().
- Underscores in names convert to hyphens by default.
- Duplicate headers are supported via list typing.
- Group headers with Pydantic models for structure and validation.
- Use Response or JSONResponse to add headers to outgoing responses.
- Mind your CORS settings when using custom headers.

Response Model and Return Type

FastAPI allows you to define what kind of data your API should return using return type annotations and the response_model parameter.

Why Define a Response Model?

- Validation: FastAPI ensures your function returns data that matches the expected structure.
- Filtering: It removes fields not defined in the response model—useful for hiding sensitive data (e.g. passwords).
- Documentation: Auto-generates clean OpenAPI docs and JSON schemas.
- Client Code Generation: Tools can auto-generate frontend clients based on these models.

Avoid Returning Sensitive Data

If input and output models are the same (e.g., a user creation API returning a user with a plain password), you risk leaking sensitive information. So, Avoid using the same model for both input and output if data visibility differs.

Form Data

When building APIs that interface with HTML forms or mobile frontends, the data is often sent in form-encoded format instead of JSON. To support this, FastAPI provides the Form utility.

Form data is submitted with content type:

- application/x-www-form-urlencoded (for fields only)
- multipart/form-data (when files are included)

Form is a subclass of Body.

If you omit Form, FastAPI assumes you're expecting JSON or query parameters.

Handling File Uploads

FastAPI allows file uploads using the File and UploadFile utilities. Files are expected via multipart/form-data.

- File() + bytes — receives the file in memory
- UploadFile — receives a file-like object, better for large files

Handling Errors

In real-world applications, APIs don't always return successful responses. Sometimes, a client might send a bad request, try to access a non-existent resource, or perform an unauthorized action. In such cases, your FastAPI application must communicate these issues clearly using proper error handling mechanisms.

Why is Error Handling Important

When developing APIs, error handling plays a crucial role in:

- Improving Client Communication: Clearly informs clients what went wrong (e.g., invalid input, resource not found).
- Maintaining Consistency: Ensures that clients always receive a predictable error structure.
- Security: Prevents leaking sensitive internal information (e.g., stack traces or unhandled exceptions).
- Debugging and Monitoring: Helps developers detect, log, and fix issues efficiently.

Error Response Format in FastAPI

FastAPI leverages standard HTTP status codes to indicate the result of a client's request:

- 2xx (Success): Indicates the request was successfully processed.
- 4xx (Client Error): The request was invalid (e.g., missing data, unauthorized).
- 5xx (Server Error): The server failed to process a valid request due to internal issues.

Raising Errors with HTTPException

To indicate an error in FastAPI, developers use the built-in HTTPException class. It allows you to:

- Set the appropriate HTTP status code.
- Attach a custom error message (called detail) which will be sent in the response.
- Optionally, add custom response headers.

This approach instantly stops the execution of the current request and sends an appropriate error response to the client.

Customizing Error Responses

FastAPI lets you register custom exception handlers, allowing you to control how specific types of errors are processed and returned. This is especially useful when:

- You want to return custom error formats.
- You want to log errors or send alerts.
- You need to support different response types (e.g., JSON, plain text).

You can handle:

- Built-in exceptions, like validation errors or HTTP status errors.
- User-defined exceptions, by creating your own exception classes and mapping them to custom responses.

Validation Errors

When incoming request data is invalid, FastAPI automatically performs validation using Pydantic models. If validation fails:

- FastAPI raises a special exception called RequestValidationError.
- The default behavior is to return a 422 Unprocessable Entity error, along with details about what failed and why.
- You can override this behavior and return custom messages, formats, or logs.

SQLAlchemy

What is SQLAlchemy?

SQLAlchemy is a Python SQL toolkit and Object Relational Mapper (ORM). In the context of FastAPI, it provides a powerful and flexible way to interact with relational databases (like MySQL, PostgreSQL, SQLite, etc.) by using Python objects instead of writing raw SQL.

Why use SQLAlchemy with FastAPI?

- Database Abstraction: SQLAlchemy abstracts away SQL queries and lets you interact with your database using Pythonic syntax.
- ORM Capabilities: It maps database tables to Python classes (known as models), allowing you to perform CRUD operations easily.
- Asynchronous Support: SQLAlchemy 2.0+ has native support for async/await, aligning well with FastAPI's asynchronous nature.
- Mature Ecosystem: It's production-ready, widely adopted, and integrates seamlessly with Alembic (for migrations), which is important in larger FastAPI projects.

Two Main Components of SQLAlchemy

- SQLAlchemy Core: A lower-level interface where you write SQL-like expressions using Python.

- SQLAlchemy ORM: A higher-level abstraction for working with databases using classes and objects. This is the part mostly used with FastAPI.

In FastAPI, we almost always use SQLAlchemy ORM.

Key Concepts in SQLAlchemy ORM

- Declarative Models - These are Python classes that represent database tables. Each attribute corresponds to a column. In FastAPI, we typically define these inside files like models.py.
- Session - A session is a database conversation handler. It manages all operations (CRUD), tracks changes to objects, and handles transactions. FastAPI often uses a dependency injection pattern to provide a session per request.
- Engine - The engine connects your app to the database. It handles the actual database dialect and connection pool. You usually define the engine in db/config.py and inject it into your session logic.
- Querying - You can query data using ORM-style expressions.
- Relationships - SQLAlchemy supports defining relationships between models (One-to-Many, Many-to-Many, etc.) using relationship().
- Migrations - SQLAlchemy itself does not handle migrations, but integrates with Alembic to manage schema changes.

Alembic: Database Migrations

What is Alembic?

Alembic is a lightweight database migration tool for use with SQLAlchemy. It allows developers to manage and version-control changes to the database schema over time in a structured and automated way.

In the context of FastAPI, Alembic ensures that database structure changes (like adding tables, altering columns, renaming fields) are applied consistently across development, staging, and production environments—without manual intervention.

Why Alembic is Important in FastAPI Projects

- Tracks Schema Evolution: Keeps a history of database changes.
- Team Collaboration: Ensures all team members share the same schema structure.
- Automation-Friendly: Integrates well with CI/CD pipelines and version control.
- Rollback Capabilities: Allows reverting problematic schema changes.
- Production-Ready: Safely manages schema changes in live databases.

Core Concepts of Alembic

Migration - A migration is a file that contains instructions for altering the database schema—adding a new table, modifying a column, or dropping an index. Alembic generates these files so the changes can be tracked and reproduced.

Revision - Every migration is versioned by a unique identifier called a revision. Revisions are linked in sequence like a chain, forming a migration history.

Head - The head is the most recent revision in the migration history. When you apply migrations, Alembic brings the database schema "up to the head."

Upgrade/Downgrade - Alembic can upgrade the database to apply newer schema changes, or downgrade it to revert back to a previous state.

Alembic Workflow in FastAPI Projects

- Define Models: You write Python models using SQLAlchemy in your FastAPI app.
- Generate Migration: Alembic scans your models and generates a migration script that reflects the schema changes.
- Apply Migration: You run the migration script to update the database schema.
- Version Tracking: Alembic records the applied revision in a special table (alembic_version) in your database.

How Alembic Syncs with SQLAlchemy

- Alembic does not read models directly from FastAPI.
- It relies on the SQLAlchemy metadata object, which must include all models to accurately detect schema changes.
- To make this work in a FastAPI project, developers often use a central model registry (e.g., models.py or base.py) that Alembic imports during migration generation.

SQL Model

What is an SQL Model?

SQLModel is:

- A library created by the author of FastAPI.
- Built on top of Pydantic (data validation) and SQLAlchemy (ORM).
- Designed to simplify the creation of models that serve both as Pydantic models and SQL tables.
- Think of it as Pydantic + SQLAlchemy with cleaner syntax.

Why use SQLModel with FastAPI?

- FastAPI already uses Pydantic for request/response validation.
- SQLModel adds SQLAlchemy ORM features to the same model class.
- This keeps your models DRY: no need to define separate Pydantic and SQLAlchemy classes.

Key SQLModel Concepts

- SQLModel base class - Base for both data and table models
- table=True – It tells SQLModel to create a table for this model
- Field() - Like Pydantic fields but also support SQL-specific metadata
- Session() - Like a DB transaction manager from SQLAlchemy
- create_engine() - Connection manager to the DB
- metadata.create_all() - Auto-creates the tables in DB using the model
- select() - SQLModel's version of querying via SQLAlchemy's Core

- model_validate() - Converts one model to another (like from input to DB model)
- model_dump() - Converts DB model to a dict
- sqlmodel_update() - Updates DB model fields dynamically

What about migrations

SQLModel doesn't (yet) have built-in migrations.

Use Alembic directly:

- It works with SQLAlchemy under the hood.
- Manage DB schema changes cleanly and track version history.
- SQL Model doesn't support async yet.

Middleware

What is Middleware?

Middleware in FastAPI is a layer of logic that:

- Intercepts every request before it reaches your API routes.
- Intercepts every response after the route handler finishes execution.
- Allows you to modify requests and responses, log events, measure time, inject headers, and more.

It sits between the client and the application logic — like a filter or interceptor.

Middleware Life Cycle

Here's the life cycle of a typical request/response in FastAPI with middleware:

Client ➞ Middleware ➞ Route Handler ➞ Middleware ➞ Client

Request Phase:

- Middleware executes before the request reaches the API route.
- You can manipulate or log the request here.

Response Phase:

- After the route returns a response, middleware runs again before the final response is sent.
- You can manipulate the response (e.g., add headers).

Common Use Cases

- Logging - Logs every request and response.
- Timing - Measures request processing time (e.g. X-Process-Time).
- Security - Validates headers or access tokens globally.
- Caching - Controls cache headers or invalidation logic.
- Compression - Automatically compresses responses (e.g. GZip).

How to Add Middleware

You can add middleware in two ways:

- Using Decorator (@app.middleware("http")) Good for quick inline middleware.
- Using app.add_middleware() for Class-based or 3rd-party Middleware

Types of Built-in Middleware

- GZipMiddleware - Compresses large responses
- HTTPSRedirectMiddleware - Redirects all HTTP traffic to HTTPS
- TrustedHostMiddleware - Prevents host header attacks
- CORSMiddleware - Enables Cross-Origin Resource Sharing
- Custom Middleware - Fully customizable request/response manipulation

Execution Order

app.add_middleware(MiddlewareA)

app.add_middleware(MiddlewareB)

- On Request: MiddlewareB → MiddlewareA → Route
- On Response: Route → MiddlewareA → MiddlewareB

API Router

When starting out with FastAPI, it's easy to place all your routes, dependencies, models, and logic in a single file. This works fine for small projects or quick experiments.

However, real-world applications—especially APIs for production, e-commerce, SaaS platforms, etc.—quickly grow too complex for a single-file structure. Without proper organization, your code becomes harder to maintain, debug, test, and scale.

That's where FastAPI's modular structure using Python packages and APIRouter becomes a game-changer.

Why use APIRouter?

- Separation of concerns: Group routes by domain (e.g., users, products, admin).
- Cleaner codebase: Keeps your main file (like main.py) minimal and readable.
- Reusability: Shared routers can be imported across different services.
- Dependency management: Apply shared dependencies at router level.

Registering Routers in the Main App

Your main.py acts as the entry point where all routers are registered using .include_router().

Tags, Prefixes, Responses

FastAPI lets you add metadata to routers and endpoints to enrich the automatic documentation:

- prefix: common route prefix (e.g., /items)
- tags: categorize routes in Swagger UI

- responses: predefine expected HTTP responses
- dependencies: enforce auth, headers, or logic across the router

Dependency Injection

Dependency Injection (DI) is a common design pattern used in software development that allows different parts of your application to depend on external logic without hard-coding the dependencies directly.

In FastAPI, this means you can write small, reusable functions (called "dependencies") and tell the framework to automatically call them when needed—passing their return values into your route handlers or other dependencies.

Why Use Dependency Injection

FastAPI's built-in dependency injection system is:

- Simple to use (just define a function and use Depends)
- Explicit and readable
- Powerful enough for advanced use cases

Here's why you might need it:

- Avoid repeating the same logic across multiple endpoints
- Centralize shared logic such as authentication and permissions
- Manage resources like database sessions or external APIs
- Enforce security rules declaratively
- Make your code easier to test and maintain

How It Works

Instead of manually calling helper functions inside your endpoints, you define these functions separately and declare them as dependencies.

FastAPI will:

- Recognize which function is needed as a dependency
- Inspect the parameters of the dependency
- Call it with the correct values from the request (e.g., query params, headers)
- Inject its return value into your endpoint or another dependency

You never call dependency functions manually. FastAPI does it for you.

Using Depends

When you declare a parameter using Depends(...), FastAPI understands that it needs to execute a function and pass its return value to your path operation.

Simplifying With Annotated Dependencies

FastAPI supports Annotated from Python's typing module, which allows cleaner dependency declarations.

You can even define type aliases to reuse complex dependency declarations across many routes—while keeping editor autocompletion and validation.

Sync vs Async Dependencies

Both async def and def functions are supported:

- FastAPI will automatically detect and handle whether a function should be awaited
- You can mix sync/async dependencies and routes freely

Nested (hierarchical) dependencies

FastAPI supports nested (hierarchical) dependencies:

For example:

- A route requires a logged-in user (current_user)
- current_user requires a valid token (token_validation)
- token_validation requires a header check (get_token_header)

FastAPI handles all of this automatically:

- Resolves each dependency in order
- Injects results step-by-step
- Reports validation errors (like missing headers or invalid tokens) immediately

This builds a tree of dependencies, where each node is another function FastAPI knows how to call.

Classes as Dependencies

While function-based dependencies are commonly used, FastAPI also supports classes as dependencies—a feature often overlooked but immensely useful for large-scale or organized applications.

Why Use Classes

- Object-oriented approach - Encapsulate related data in a clean class structure.
- IDE support - Your editor can infer available properties, offer autocompletion, and perform static checks.
- Reusability - The same class can be reused in multiple routes or services.
- Cleaner logic separation - Class initialization (__init__) can hold logic, keeping endpoints minimal.

Sub-dependencies

A sub-dependency is simply a dependency that relies on another dependency.

FastAPI allows you to declare dependencies that themselves declare other dependencies. This results in a dependency tree, and FastAPI automatically resolves this tree by:

- Analyzing all declared dependencies and their sub-dependencies.
- Executing each dependency in the correct order.
- Caching results for performance unless specified otherwise.

You can nest these as deep as needed—FastAPI handles all the complexity.

Global Dependencies

Global dependencies are dependencies that are declared at the FastAPI application level, not just for individual routes or routers.

This means:

- Every path operation (endpoint) will automatically depend on them.
- You don't need to repeat the same dependency in each endpoint.
- They apply application-wide logic consistently.

Dependencies with yield

In FastAPI, dependencies can do more than just provide a value. They can also execute cleanup logic after the response is created—by using the yield keyword instead of return.

This is especially useful for managing resources like database connections, file handles, or external service connections that must be properly closed or cleaned up once the request is complete.

Why yield Instead of return

When a dependency uses yield, FastAPI:

- Executes code before yield → to prepare resources.
- Provides the yielded value → to the path operation or another dependency.
- Executes code after yield → for cleanup or shutdown once the response is created (but before it is sent).

This mimics the behavior of Python's context managers (with statements) and is powered internally by contextlib.contextmanager or asynccontextmanager.

Background Tasks

Background tasks allow you to execute certain operations after a response has been sent to the client. This pattern is extremely useful when you want to:

- Improve performance from the user's perspective.
- Defer slow or non-critical operations.
- Keep the request/response cycle fast and efficient.

How FastAPI Handles It

FastAPI provides a class called BackgroundTasks, which is built on top of Starlette's background system. This class:

- Collects one or more tasks.
- Executes them after the response is sent.
- Is automatically injected by FastAPI when declared as a parameter.

FastAPI handles both normal functions and async functions correctly under the hood.

Background Task Execution Flow

- The client makes a request.
- FastAPI runs the path operation function.
- Inside this function, background tasks are registered via .add_task().
- The response is returned to the client.
- Then, background tasks are executed after the response is sent.

This flow ensures that users don't experience any delay caused by post-response logic.

WebSockets

WebSockets provide a persistent, bidirectional communication channel between the client and server. Unlike HTTP, which is request-response based, WebSockets allow real-time communication, making them ideal for applications like chat systems, notifications, gaming, and live updates.

Why Use WebSockets

FastAPI, being built on Starlette, natively supports asynchronous WebSocket endpoints. This makes FastAPI an ideal choice when combining REST APIs with real-time features.

Key Benefits:

- Integrated support using WebSocket from fastapi.
- Seamless use of FastAPI's dependency injection.
- Can use query parameters, cookies, headers, etc., even in WebSocket routes.

Core Concepts

- WebSocket object - Represents the connection, used to accept, send, and receive messages.
- @app.websocket() - Decorator to define a WebSocket endpoint.
- receive_text() - Awaits and receives incoming messages as strings.
- send_text() - Sends messages back to the client.
- WebSocketDisconnect - Exception raised when a client disconnects.

Serving Static Files

In web development, static files refer to resources like images, JavaScript, CSS, fonts, or HTML that do not change dynamically per request. These are served as-is to the client, without server-side rendering or processing.

FastAPI provides a convenient way to serve such static content using StaticFiles.

Why Serve Static Files?

Static files are essential for:

- Frontend assets in full-stack applications.
- Documentation or user guides (in HTML).
- Embedded files like logos, PDFs, or fonts.

- Hosting simple frontend websites alongside APIs.

How FastAPI Handles Static Files

FastAPI enables static file serving through the StaticFiles class, which is built on top of Starlette, the ASGI toolkit FastAPI is based on.

To serve static files:

- Import StaticFiles.
- Mount it using app.mount() on a specific path.
- Specify the directory from which files will be served.

Mounting

Mounting refers to the process of attaching a separate application or handler (like StaticFiles) to a specific sub-path of your main FastAPI application.

- This mounted handler is independent from the main app.
- It controls all traffic under its path (e.g., /static/...).
- The OpenAPI docs do not include routes from mounted apps.

Think of mounting as delegating a sub-path of your application to another mini-application specialized in serving static files.

Templates

In modern web applications, it's common to return HTML pages instead of (or alongside) JSON responses. While FastAPI is optimized for APIs, it also supports rendering HTML templates, just like traditional web frameworks such as Flask or Django.

Why Use Templates?

Templates are useful when:

- Building server-side rendered pages (e.g., admin dashboards, login forms).
- Generating dynamic HTML responses.
- Reusing layout logic (header, footer, menus) through inheritance.
- Mixing API endpoints and frontend rendering within the same project.

FastAPI allows seamless integration with popular template engines, especially Jinja2, for building dynamic web pages.

Template Engine Support

FastAPI supports any template engine, but the most commonly used is Jinja2. It's a powerful, Pythonic templating system used in Flask and other Python frameworks.

The integration with Jinja2 is made simple through utilities provided by Starlette, which FastAPI builds upon.

How Template Rendering Works

To render an HTML template:

- Install Jinja2 via pip install jinja2.
- Import and configure Jinja2Templates by specifying a template directory.
- Mount static files (CSS, JS, images) using StaticFiles.
- Use TemplateResponse to return an HTML response to the client.
- Pass a context dictionary containing variables to render inside the HTML.

Note:-

- HTML Templating is synchronous. It doesn't take full advantage of FastAPI's async capabilities.
- Templates should be used only when you need server-side rendering. Otherwise, prefer frontend frameworks or API-based architecture.
- Avoid logic-heavy templates — keep business logic in Python, not HTML.

Metadata and Documentation

FastAPI comes with powerful support for generating documentation automatically from your code using the OpenAPI specification. These docs are not only helpful for developers but also crucial for communicating API design, usage, and intent in professional environments.

You can enrich this documentation with metadata, organize endpoints using tags, and even customize the docs URL paths for branding or versioning purposes.

Why Metadata Matters

Metadata improves the understandability, discoverability, and documentation quality of your API. Tools like Swagger UI and ReDoc rely on this metadata to auto-generate interactive documentation.

This is especially useful in:

- Public APIs that require consumer clarity
- Internal microservices that need organized communication
- Interview settings where API design is assessed

FastAPI Metadata Fields

When creating a FastAPI app, you can provide optional parameters to customize metadata:

Parameter	Type	Purpose
Title	str	Main title of your API.
Summary	str	Short summary of your API (OpenAPI 3.1.0+).
Description	str	Detailed markdown-supported description of the API.
Version	str	Current version of your application.
Terms_of_service	str	URL pointing to your terms of service.
Contact	dict	Author or team contact information (name, email, URL).
License_info	dict	License details including name, URL or identifier.

Testing

Testing is a critical part of building reliable web applications. In FastAPI, testing is intuitive, fast, and simple thanks to its foundation on Starlette and the use of HTTPX. It supports both unit testing and integration testing out-of-the-box.

TestClient

FastAPI provides TestClient (from fastapi.testclient) which internally uses Starlette's TestClient, which is based on HTTPX. It allows you to simulate HTTP requests to your app just like a real client (e.g., Postman or a browser).

- You don't need to start a real server.
- Tests run synchronously using normal def functions (not async def).
- Designed to work seamlessly with pytest.