FastAPI Fundamentals

1. What Is FastAPI?

FastAPI is a modern Python web framework for building APIs.

It is designed around Python type hints and provides features such as:

- HTTP routing
- Request validation
- JSON handling
- Automatic API documentation
- Dependency injection
- Async support
- OpenAPI schema generation

FastAPI uses Starlette for its web functionality and Pydantic for data validation and serialization.

Simple mental model

```
Client
   ↓
HTTP Request
   ↓
FastAPI
   ↓
Path Operation
   ↓
Python Function
   ↓
Business Logic
   ↓
JSON Response
   ↓
Client
```

For example:

```
GET /api/resources
        ↓
FastAPI finds the matching route
        ↓
Python function executes
        ↓
```

```
Data is returned
        ↓
FastAPI converts it to JSON
```

---

2. Why FastAPI?

Python has several web frameworks.

For example:

```
Flask
Django
FastAPI
```

FastAPI focuses heavily on API development.

One of its important ideas is:

> Use Python type hints as part of your API definition.

For example:

```
@app.get("/items/{item_id}")
def get_item(item_id: int):
    ...
```

The `int` tells FastAPI that `item_id` should be an integer. FastAPI uses that declaration for validation and OpenAPI documentation.

This means your Python code is doing more than simply documenting the expected type.

It helps define the API contract.

---

3. Flask vs FastAPI

You have already learned Flask.

The basic idea is similar:

```
Client
   ↓
HTTP
   ↓
Python backend
   ↓
Response
```

But the developer experience differs.

Flask

```
from flask import Flask

app = Flask(__name__)


@app.get("/items/<int:item_id>")
def get_item(item_id):
    return {
        "item_id": item_id
    }
```

FastAPI

```
from fastapi import FastAPI

app = FastAPI()


@app.get("/items/{item_id}")
def get_item(item_id: int):
    return {
        "item_id": item_id
    }
```

FastAPI can use the type annotation to validate the incoming value and include it in the generated API schema.

---

4. Install FastAPI

Create a project:

```
fastapi-project/
```

Create a virtual environment:

```
python -m venv .venv
```

Activate it.

Windows

```
.venv\Scripts\activate
```

macOS/Linux

```
source .venv/bin/activate
```

Install FastAPI:

```
pip install "fastapi[standard]"
```

The official documentation also supports installing FastAPI with `uv`, but using `pip` works well for understanding traditional Python project setup.

---

5. Your First FastAPI Application

Create:

```
main.py
```

Add:

```
from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def root():
    return {
        "message": "Hello, FastAPI!"
    }
```

Run the development server:

```
fastapi dev
```

FastAPI's current development workflow supports `fastapi dev`, which starts the development server and reloads the application as appropriate for development.

Open:

```
http://127.0.0.1:8000/
```

You should receive:

```
{
  "message": "Hello, FastAPI!"
}
```

---

6. What Is `FastAPI()`?

This:

```
app = FastAPI()
```

creates the FastAPI application object.

It becomes the central object through which you define:

- Routes
- API metadata
- Middleware
- Dependencies
- Exception handlers
- Configuration

Think of:

```
app = FastAPI()
```

as creating your API application.

---

7. Path Operations

A FastAPI endpoint is commonly called a **path operation**.

For example:

```
@app.get("/users")
def get_users():
    return []
```

There are two important pieces:

```
@app.get("/users")
       ↓
Path + HTTP method
```

and:

```
def get_users():
```

the function that handles the request.

FastAPI calls the function when the matching HTTP request arrives.

---

8. HTTP Methods

FastAPI provides decorators for common HTTP methods:

```
@app.get(...)
@app.post(...)
@app.put(...)
@app.patch(...)
@app.delete(...)
```

Common API meaning:

Method	**Typical purpose**
GET	Read
POST	Create
PUT	Replace/update
PATCH	Partial update
DELETE	Delete

For example:

```
@app.get("/resources")
def get_resources():
    ...


@app.post("/resources")
def create_resource():
    ...


@app.put("/resources/{resource_id}")
def update_resource(resource_id: int):
    ...


@app.delete("/resources/{resource_id}")
```

```
def delete_resource(resource_id: int):
    ...
```

These are called path operations in FastAPI.

---

9. Returning JSON

You can return a Python dictionary:

```
@app.get("/status")
def status():
    return {
        "status": "ok"
    }
```

FastAPI converts the result into a JSON response.

You can also return lists:

```
@app.get("/tags")
def get_tags():
    return [
        "python",
        "fastapi",
        "api"
    ]
```

FastAPI supports returning Python data such as dictionaries, lists, strings, numbers, and Pydantic models and converts them appropriately for the response.

---

10. Path Parameters

Suppose we want:

```
GET /resources/10
```

Define:

```
@app.get("/resources/{resource_id}")
def get_resource(resource_id: int):
    return {
        "id": resource_id
    }
```

The `{resource_id}` is a path parameter.

The type:

```
resource_id: int
```

tells FastAPI to expect an integer.

Request:

```
/resources/10
```

works.

But:

```
/resources/hello
```

does not satisfy the declared integer type.

FastAPI performs the validation and generates the corresponding API schema automatically.

---

11. Query Parameters

Query parameters appear after `?`.

For example:

```
/resources?category=python
```

FastAPI can declare them directly in the function:

```
@app.get("/resources")
def get_resources(category: str | None = None):
    return {
        "category": category
    }
```

Request:

```
/resources?category=python
```

produces:

```
{
  "category": "python"
}
```

Without the parameter:

```
/resources
```

the value is:

```
None
```

FastAPI determines parameters such as path and query values from the function signature and their type declarations.

---

12. Required Query Parameters

You can make a parameter required:

```
@app.get("/search")
def search(q: str):
    return {
        "query": q
    }
```

The client must provide:

```
/search?q=python
```

If `q` is missing, FastAPI returns a validation error.

---

13. Optional Query Parameters

Use a default value:

```
@app.get("/search")
def search(q: str | None = None):
    return {
        "query": q
    }
```

Now both work:

```
/search
```

and:

```
/search?q=python
```

This is one of the advantages of using Python's type annotations directly in FastAPI.

---

14. Multiple Query Parameters

You can declare several:

```
@app.get("/resources")
def get_resources(
    category: str | None = None,
    level: str | None = None,
    limit: int = 10
):
    return {
        "category": category,
        "level": level,
        "limit": limit
    }
```

Request:

```
/resources?category=python&level=beginner&limit=5
```

FastAPI parses these values according to the declared types.

---

15. Request Bodies

A POST request often sends JSON data.

For example:

```
{
  "title": "FastAPI Fundamentals",
  "level": "Beginner"
}
```

Instead of manually reading and validating every field, FastAPI commonly uses Pydantic models.

---

16. Pydantic Models

Create a model:

```
from pydantic import BaseModel


class Resource(BaseModel):
    title: str
    level: str
```

Then:

```
@app.post("/resources")
def create_resource(resource: Resource):
    return resource
```

Now FastAPI knows the expected request body structure.

Example:

```
{
  "title": "FastAPI Fundamentals",
  "level": "Beginner"
}
```

FastAPI reads the JSON body, converts it into the declared model, validates it, and includes the model's schema in the automatically generated OpenAPI documentation.

---

17. Why Pydantic Is Important

Without a validation model, you might write:

```
data = request.json

title = data["title"]
level = data["level"]
```

and manually check everything.

FastAPI lets you declare:

```
class Resource(BaseModel):
    title: str
    level: str
```

The framework can then validate incoming data according to that model.

Think:

```
JSON
 ↓
Pydantic model
 ↓
Validation
 ↓
Python object
 ↓
Your function
```

---

18. Validation Example

Consider:

```
class Resource(BaseModel):
    title: str
    price: float
```

Valid:

```
{
  "title": "Python Guide",
  "price": 10.5
}
```

Invalid data such as a missing required field will result in a validation error before your path operation receives the model.

FastAPI's validation errors are designed to identify what part of the incoming data was invalid.

---

19. Optional Fields

You can define:

```
class Resource(BaseModel):
    title: str
    description: str | None = None
```

Now:

```
{
  "title": "FastAPI Fundamentals"
}
```

is valid.

The description is optional.

---

20. Nested Models

Models can contain other models.

```
class Author(BaseModel):
    name: str
```

```
class Resource(BaseModel):
    title: str
    author: Author
```

Request:

```
{
  "title": "FastAPI Fundamentals",
  "author": {
    "name": "Hafsa"
  }
}
```

FastAPI and Pydantic can validate nested JSON structures as well.

---

21. Response Models

Request validation is only half the problem.

You should also control what your API returns.

Example:

```
class Resource(BaseModel):
    title: str
    level: str


@app.get("/resources/{resource_id}", response_model=Resource)
def get_resource(resource_id: int):
    return {
        "title": "FastAPI Fundamentals",
        "level": "Beginner",
        "internal_note": "not public"
    }
```

The response model defines the public response structure.

This helps ensure that only the fields intended for the API response are returned. FastAPI's `response_model` is also used for OpenAPI documentation, serialization, and filtering response fields.

---

22. Request Model vs Response Model

Do not automatically use one model for everything.

For example:

```
class ResourceCreate(BaseModel):
    title: str
    description: str


class ResourceResponse(BaseModel):
    id: int
    title: str
    description: str
```

The client sends:

```
{
  "title": "FastAPI",
  "description": "Learn FastAPI"
}
```

The server returns:

```
{
  "id": 1,
  "title": "FastAPI",
  "description": "Learn FastAPI"
}
```

The database-generated `id` does not need to be provided by the client.

---

23. Status Codes

FastAPI lets you declare response status codes.

```
from fastapi import FastAPI, status


@app.post(
    "/resources",
    status_code=status.HTTP_201_CREATED
)
def create_resource(resource: Resource):
    return resource
```

A successful creation now returns:

```
201 Created
```

FastAPI includes declared status codes in its OpenAPI schema and documentation.

---

24. Raising HTTP Errors

Suppose a resource doesn't exist.

Use:

```
from fastapi import HTTPException


@app.get("/resources/{resource_id}")
def get_resource(resource_id: int):

    if resource_id != 1:
        raise HTTPException(
            status_code=404,
            detail="Resource not found"
        )

    return {
        "id": 1,
        "title": "FastAPI Fundamentals"
    }
```

The client receives an error response instead of a successful response.

---

25. Automatic API Documentation

One of FastAPI's major features is automatic interactive documentation.

After starting the application:

```
http://127.0.0.1:8000/docs
```

opens Swagger UI.

You can:

- See endpoints
- See HTTP methods
- See parameters
- See request schemas
- See response schemas

- Send requests directly
- Inspect responses

FastAPI also provides:

```
http://127.0.0.1:8000/redoc
```

for ReDoc.

FastAPI automatically generates an OpenAPI schema that powers these documentation interfaces.

---

26. OpenAPI

OpenAPI is a standard way to describe an API.

FastAPI generates this description automatically.

The schema describes things such as:

```
Paths
Methods
Parameters
Request bodies
Responses
Schemas
Status codes
```

For example:

```
GET /resources
POST /resources
GET /resources/{resource_id}
```

can all be represented in the generated OpenAPI schema.

This is valuable because other tools can understand the API contract.

---

27. API Documentation From Python Types

Consider:

```
class Resource(BaseModel):
    title: str
    level: str


@app.post("/resources")
```

```
def create_resource(resource: Resource):
    return resource
```

From this small amount of code, FastAPI can generate information about:

```
POST /resources

Request body:
Resource

title:
string

level:
string
```

This is a major part of the FastAPI developer experience.

---

28. Tags

You can organize documentation with tags:

```
@app.get(
    "/resources",
    tags=["resources"]
)
def get_resources():
    return []
```

And:

```
@app.get(
    "/users",
    tags=["users"]
)
def get_users():
    return []
```

Swagger UI groups the endpoints accordingly.

FastAPI supports tags, summaries, descriptions, response descriptions, and other path-operation metadata for generated documentation.

---

29. Async Functions

FastAPI supports asynchronous path operations:

```
@app.get("/resources")
async def get_resources():
    return []
```

You can also use a normal function:

```
@app.get("/resources")
def get_resources():
    return []
```

Both are valid.

Use `async def` when working with asynchronous operations and libraries.

For example, an async database or HTTP client can be used without blocking the event loop in the same way a synchronous call would.

FastAPI's documentation explicitly supports both regular `def` and `async def` path operation functions.

---

30. Why Async Matters

Imagine your server needs to wait for an external API:

```
Your API
   ↓
External API
   ↓
Wait...
   ↓
Response
```

With asynchronous programming, the server can handle other suitable work while waiting for asynchronous I/O.

The important point is:

> `async` is primarily about efficiently handling waiting, especially I/O-bound work.

It does not automatically make CPU-heavy code faster.

---

31. Dependency Injection

FastAPI includes a dependency injection system.

Example:

```
from fastapi import Depends


def common_parameters():
    return {
        "limit": 10
    }


@app.get("/resources")
def get_resources(
    params = Depends(common_parameters)
):
    return params
```

FastAPI calls the dependency and provides its result to the path operation.

Dependencies can be used for:

- Authentication
- Authorization
- Database sessions
- Shared query parameters
- Common validation
- Reusable services

FastAPI's dependency system lets the framework execute required dependencies and inject their results into path operations.

---

32. Why Dependency Injection Matters

Without dependency injection, you might repeat:

```
database = create_database_connection()
```

in many endpoints.

With dependencies:

```
Endpoint
   ↓
Dependency
   ↓
Database session
```

The endpoint simply declares what it needs.

This improves:

- Reusability
- Testing
- Separation of concerns
- Maintainability

---

33. Organizing a FastAPI Project

A tiny project can start with:

```
fastapi-app/
│
├── main.py
├── requirements.txt
└── .venv/
```

As it grows:

```
fastapi-app/
│
├── app/
│   ├── main.py
│   │
│   ├── routers/
│   │   ├── resources.py
│   │   ├── users.py
│   │   └── auth.py
│   │
│   ├── schemas/
│   │   ├── resource.py
│   │   └── user.py
│   │
│   ├── services/
│   │   └── resource_service.py
│   │
│   ├── models/
│   │   └── resource.py
│   │
│   └── dependencies.py
│
```

```
├── tests/
│
├── requirements.txt
└── .env
```

The exact architecture depends on the size and requirements of the project.

---

34. Routers

FastAPI provides `APIRouter` for organizing routes.

```
from fastapi import APIRouter

router = APIRouter()


@router.get("/resources")
def get_resources():
    return []
```

Then register it in the application:

```
from fastapi import FastAPI

from app.routers import resources

app = FastAPI()

app.include_router(resources.router)
```

Now resource routes can live in their own module.

---

35. Separating Responsibilities

Avoid putting everything inside a route.

Instead of:

```
@app.post("/resources")
def create_resource(resource: Resource):
    # validate
    # calculate
    # database query
    # business rules
```

```
    # send notification
    # return response
```

prefer:

```
Router
  ↓
Service
  ↓
Repository
  ↓
Database
```

For example:

```
@router.post("/resources")
def create_resource(resource: ResourceCreate):
    return resource_service.create(resource)
```

The service contains business logic.

This makes the code easier to test and maintain.

---

36. Database Integration

A real API normally needs persistent storage.

The architecture might become:

```
Client
   ↓
FastAPI
   ↓
Router
   ↓
Service
   ↓
Repository / ORM
   ↓
PostgreSQL
```

Possible databases include:

```
SQLite
PostgreSQL
MySQL
```

Database integration introduces additional concepts such as:

- Models
- Queries
- Relationships
- Transactions
- Migrations
- Connection management

Those should be learned separately rather than treating the in-memory list as a real database.

---

37. Environment Variables

Production applications should not hard-code secrets.

Bad:

```
DATABASE_PASSWORD = "my-secret-password"
```

Better:

```
DATABASE_URL
SECRET_KEY
API_KEY
```

stored in the environment.

Then Python can read them using:

```
import os

database_url = os.getenv("DATABASE_URL")
```

For local development, a `.env` file can be used with an appropriate environment configuration library.

---

38. Authentication and Authorization

A production API often needs users.

Two separate concepts:

Authentication

```
Who are you?
```

Authorization

```
What are you allowed to do?
```

For example:

```
User
 ↓
Login
 ↓
Authentication
 ↓
Token
 ↓
Request
 ↓
Authorization
 ↓
Access resource
```

FastAPI provides security utilities and dependency injection that can be combined to implement authentication and authorization.

---

39. CORS

Suppose:

```
Frontend:
https://example.com

API:
https://api.example.com
```

The frontend and API may require Cross-Origin Resource Sharing configuration.

FastAPI applications can use CORS middleware to control which origins are allowed.

Do not blindly allow every origin in production.

Prefer explicitly configured trusted origins.

---

40. Pagination

Suppose your API has:

```
500,000 resources
```

Do not return all of them:

```
GET /resources
```

Instead:

```
GET /resources?skip=0&limit=20
```

or:

```
GET /resources?page=1&limit=20
```

The exact pagination design depends on the application.

The principle is:

> Return a manageable amount of data per request.

---

41. Filtering and Searching

An endpoint might support:

```
GET /resources?category=python
```

or:

```
GET /resources?level=beginner
```

or:

```
GET /resources?search=fastapi
```

Multiple filters can be combined:

```
GET /resources?category=python&level=beginner
```

FastAPI's typed query parameters make these declarations straightforward.

---

42. Response Models and Security

Response models are not just documentation.

Suppose your internal object contains:

```
{
    "id": 1,
    "name": "Hafsa",
    "email": "...",
    "password_hash": "..."
}
```

But your response model is:

```
class UserResponse(BaseModel):
    id: int
    name: str
```

Then the public response can expose only the fields defined by that model.

FastAPI's response-model processing can filter returned data to the declared response shape.

This is useful for preventing accidental exposure of internal fields.

---

43. Error Handling

Use:

```
from fastapi import HTTPException
```

Example:

```
if resource is None:
    raise HTTPException(
        status_code=404,
        detail="Resource not found"
    )
```

Common API status codes:

Code	Meaning
200	Success
201	Created
204	Success with no body
400	Bad request
401	Not authenticated

403	Not authorized
404	Not found
409	Conflict
422	Validation error
500	Server error

FastAPI automatically performs validation for declared request data and returns validation errors when input doesn't satisfy the declared types or models.

---

44. A Complete Small FastAPI Example

Here is a compact resource API:

```
from fastapi import FastAPI, HTTPException, status
from pydantic import BaseModel

app = FastAPI()


class ResourceCreate(BaseModel):
    title: str
    category: str


class Resource(ResourceCreate):
    id: int


resources = [
    Resource(
        id=1,
        title="Python Basics",
        category="Python"
    ),
```

```
    Resource(
        id=2,
        title="FastAPI Fundamentals",
        category="Python"
    )
]


@app.get("/")
def root():
    return {
        "message": "FastAPI is running"
    }


@app.get(
    "/resources",
    response_model=list[Resource]
)
def get_resources(
    category: str | None = None
):
    if category:
        return [
            resource
            for resource in resources
            if resource.category.lower() == category.lower()
        ]

    return resources


@app.get(
    "/resources/{resource_id}",
    response_model=Resource
)
def get_resource(resource_id: int):

    for resource in resources:
        if resource.id == resource_id:
            return resource

    raise HTTPException(
```

```
        status_code=404,
        detail="Resource not found"
    )


@app.post(
    "/resources",
    response_model=Resource,
    status_code=status.HTTP_201_CREATED
)
def create_resource(resource: ResourceCreate):

    new_resource = Resource(
        id=len(resources) + 1,
        **resource.model_dump()
    )

    resources.append(new_resource)

    return new_resource


@app.delete("/resources/{resource_id}")
def delete_resource(resource_id: int):

    for index, resource in enumerate(resources):
        if resource.id == resource_id:
            resources.pop(index)

            return {
                "message": "Resource deleted"
            }

    raise HTTPException(
        status_code=404,
        detail="Resource not found"
    )
```

Notice how much information FastAPI gets from the Python types:

```
resource_id: int
```

```
resource: ResourceCreate
```

```
response_model=list[Resource]
```

These declarations contribute to validation, serialization, and generated API documentation.

---

45. Testing FastAPI

FastAPI applications should have automated tests.

A typical test might use FastAPI's testing support with pytest.

Conceptually:

```
def test_get_resources(client):
    response = client.get("/resources")

    assert response.status_code == 200
```

You should test:

```
Successful GET
Successful POST
Invalid POST
Missing resource
Invalid path parameter
DELETE
Filtering
Authentication
Authorization
```

Testing should verify both successful behavior and failure behavior.

---

46. FastAPI Development Workflow

A practical workflow:

```
1. Define API resource
        ↓
2. Define Pydantic schemas
        ↓
3. Define route
        ↓
4. Validate input
        ↓
5. Execute business logic
        ↓
```

```
6. Access database/service
        ↓
7. Define response model
        ↓
8. Test endpoint
        ↓
9. Check /docs
        ↓
10. Deploy
```

This creates a repeatable development process.

---

47. FastAPI + Frontend

A frontend can communicate with FastAPI:

```
React / Next.js
       ↓
    fetch()
       ↓
HTTP Request
       ↓
   FastAPI
       ↓
Database
       ↓
JSON
       ↓
Frontend
```

Example JavaScript:

```
const response = await fetch(
  "http://localhost:8000/resources"
);

const resources = await response.json();

console.log(resources);
```

FastAPI does not replace the frontend.

It provides the backend API.

---

48. FastAPI + Mobile Apps

The same API can serve a mobile application.

```
                ┌── Web App
                │
FastAPI API ────┼── Mobile App
                │
                └── Admin Dashboard
```

This is one reason APIs are valuable.

The clients can change while the backend contract remains stable.

---

49. Real-World haas.dev Example

Imagine haas.dev has a resource system.

The frontend requests:

```
GET /api/resources?category=Python
```

FastAPI route:

```
@app.get("/api/resources")
def get_resources(
    category: str | None = None
):
    return resource_service.list(
        category=category
    )
```

A Pydantic response model could define:

```
class ResourceResponse(BaseModel):
    id: int
    title: str
    category: str
    level: str
```

The client receives predictable JSON:

```
[
  {
    "id": 1,
    "title": "Python Basics",
```

```
    "category": "Python",
    "level": "Beginner"
  }
]
```

FastAPI can then expose this endpoint automatically through `/docs`.

That means the API contract becomes visible and testable without manually writing a separate API documentation page.

---

50. Common Mistakes

Mistake 1: Treating type hints as comments

This:

```
resource_id: int
```

is important.

FastAPI uses the declaration for validation and API schema generation.

Mistake 2: Returning database objects without thinking about the response

Define response models when you need control over the public API shape.

Mistake 3: Putting all logic in `main.py`

A small learning project can use one file.

A larger application should use routers, services, schemas, models, and dependencies.

Mistake 4: Ignoring validation

Never trust:

```
Browser
Mobile app
External API
User
```

to send perfect data.

Validate at the API boundary.

Mistake 5: Confusing authentication with authorization

```
Authentication → identity
Authorization → permissions
```

Mistake 6: Making everything asynchronous

`async` is useful when your operations are asynchronous and I/O-bound.

It is not a magic performance switch.

Mistake 7: Returning huge datasets

Use:

```
Pagination
Filtering
Searching
```

instead.

Mistake 8: Exposing internal fields

Use response models to control what leaves the API.

---

51. Five-Minute Challenge

Create a FastAPI endpoint:

```
GET /api/health
```

It should return:

```
{
  "status": "healthy",
  "service": "resource-api"
}
```

Then add:

```
GET /api/resources/{resource_id}
```

Requirements:

- `resource_id` must be an integer.
- Return a resource if it exists.
- Return `404` if it doesn't.
- Add a response model.

- Check the endpoint in `/docs`.

---

52. Exercises

Exercise 1: Create a User Model

Create:

```
class User(BaseModel):
    username: str
    email: str
```

Then build:

```
POST /users
```

Exercise 2: Add Filtering

Build:

```
GET /resources?category=Python
```

Exercise 3: Add Pagination

Support:

```
GET /resources?skip=0&limit=10
```

Exercise 4: Add PATCH

Create:

```
PATCH /resources/{resource_id}
```

Allow the client to update only selected fields.

Exercise 5: Add a Dependency

Create a reusable dependency that provides:

```
{
    "limit": 10
}
```

Use it in two endpoints.

---

53. Mini Project: FastAPI Resource API

Build a backend for a learning-resource platform.

Resource

```
id
title
description
category
level
```

Endpoints

```
GET    /api/resources
GET    /api/resources/{id}
POST   /api/resources
PUT    /api/resources/{id}
PATCH  /api/resources/{id}
DELETE /api/resources/{id}
```

Add:

- Pydantic request models
- Pydantic response models
- Input validation
- Query filtering
- Pagination
- HTTP error handling
- API documentation
- Automated tests
- Router separation

Then replace the in-memory list with a real database.

---

54. Interview Questions

Beginner

1. What is FastAPI?
2. What is a path operation?
3. What does `FastAPI()` create?
4. What is a path parameter?
5. What is a query parameter?

6. What is Pydantic?
7. What is a request body?
8. What is `HTTPException`?
9. What is OpenAPI?
10. Where can you find FastAPI's interactive documentation?

Intermediate

11. Why does FastAPI use Python type hints?
12. What is the purpose of `response_model`?
13. What is the difference between request and response models?
14. What are FastAPI dependencies?
15. What is `APIRouter`?
16. What is the difference between `def` and `async def` in FastAPI?
17. How does FastAPI validate request bodies?
18. How does FastAPI generate API documentation?
19. Why should large applications separate routers and services?
20. Why is response validation useful?

Advanced

21. How would you structure a production FastAPI application?
22. How would you implement authentication with dependencies?
23. How would you manage database sessions?
24. How would you design pagination?
25. How would you prevent sensitive fields from being returned?
26. How would you test dependencies?
27. How would you handle background work?
28. How would you deploy a FastAPI application?
29. How would you monitor a production API?
30. When would asynchronous programming provide a benefit?

---

55. FastAPI Cheat Sheet

Create application

```
from fastapi import FastAPI

app = FastAPI()
```

GET

```
@app.get("/resources")
def get_resources():
    return []
```

POST

```
@app.post("/resources")
def create_resource():
    ...
```

Path parameter

```
@app.get("/resources/{resource_id}")
def get_resource(resource_id: int):
    ...
```

Query parameter

```
@app.get("/resources")
def get_resources(
    category: str | None = None
):
    ...
```

Pydantic model

```
from pydantic import BaseModel


class Resource(BaseModel):
    title: str
    category: str
```

Request body

```
@app.post("/resources")
def create_resource(resource: Resource):
    return resource
```

Response model

```
@app.get(
    "/resources/{resource_id}",
```

```
    response_model=Resource
)
def get_resource(resource_id: int):
    ...
```

Status code

```
from fastapi import status

@app.post(
    "/resources",
    status_code=status.HTTP_201_CREATED
)
def create_resource():
    ...
```

HTTP error

```
from fastapi import HTTPException

raise HTTPException(
    status_code=404,
    detail="Resource not found"
)
```

Dependency

```
from fastapi import Depends

def get_settings():
    return {"limit": 10}


@app.get("/resources")
def get_resources(
    settings=Depends(get_settings)
):
    ...
```

Router

```
from fastapi import APIRouter

router = APIRouter()
```

```
@router.get("/resources")
def get_resources():
    return []
```

Documentation

```
/docs
```

Swagger UI:

```
http://127.0.0.1:8000/docs
```

ReDoc:

```
http://127.0.0.1:8000/redoc
```

FastAPI generates both interfaces from its OpenAPI schema.

---

56. Flask vs FastAPI Mental Model

You should not think:

```
Flask = old
FastAPI = new
```

Instead:

```
Flask
  ↓
Flexible Python web framework
  ↓
You choose more of the structure

FastAPI
  ↓
API-focused Python framework
  ↓
Type hints + validation + OpenAPI
```

Both can build APIs.

The important difference for learning is that FastAPI makes API contracts, validation, schemas, and documentation a much more integrated part of the development experience.

---

57. Key Takeaways

- FastAPI is a Python framework focused heavily on API development.
- Routes in FastAPI are called path operations.
- HTTP methods such as GET, POST, PUT, PATCH, and DELETE map naturally to API operations.
- Python type hints are an important part of FastAPI.
- Path parameters can be declared directly in function parameters.
- Query parameters can be declared directly in function signatures.
- Pydantic models define and validate structured request data.
- Response models define the public response shape.
- FastAPI automatically generates OpenAPI documentation.
- `/docs` provides interactive Swagger UI.
- `/redoc` provides an alternative documentation interface.
- `HTTPException` is useful for controlled API errors.
- Dependency injection provides reusable application components.
- `APIRouter` helps organize larger applications.
- `async def` is useful for asynchronous I/O but should not be treated as a universal performance solution.
- Real applications should separate routes, business logic, data access, and schemas.
- Production APIs need authentication, authorization, validation, security, testing, monitoring, and persistent storage.
- FastAPI's biggest learning advantage is the close relationship between **Python types, validation, API contracts, and documentation**.

Core mental model

```
Python Type Hints
       ↓
FastAPI
       ↓
Validation
       ↓
Path Operation
       ↓
Business Logic
       ↓
Response Model
       ↓
```

```
JSON
       ↓
OpenAPI Documentation
```

Remember:

> In FastAPI, your type declarations are not just hints for the programmer. They can become part of the API's validation, serialization, and documented contract.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app