✅ 1. FastAPI App Setup

```python
from fastapi import FastAPI
app = FastAPI()
```

➤ What is `fastapi`?

- `FastAPI` is a **modern, high-performance Python web framework** for building APIs.
- It's based on **Starlette** for the web parts and **Pydantic** for data validation and type declarations.
- Unlike Flask or Django, FastAPI uses **Python type hints** to automatically:
  - Validate request data.
  - Generate docs (Swagger UI, Redoc).
  - Improve editor autocomplete and error checking.

➤ What is `FastAPI`?

- `FastAPI` (with capital `F`) is a **class** inside the `fastapi` module.
- When you do `FastAPI()`, you're **creating an instance** of this class.
- This instance is the **main application object**, similar to how you'd create an app in Flask:

  ```python
  from flask import Flask
  app = Flask(__name__)
  ```

🔹 `app = FastAPI()`

➤ What's happening here?

- You're **instantiating the FastAPI class**, meaning you're creating an **object** named `app`.
- `app` is your **application instance**. You will use it to:
  - Define API endpoints (routes).
  - Add middleware.
  - Customize metadata like title, version, description, etc.
  - Run the app using `uvicorn`.

➤ Internally:

When `FastAPI()` is called:

- It **inherits** from `Starlette` (a powerful async web framework).
- It sets up:
    - Routing system (`/users`, `/items`, etc.)
    - Request handling
    - Automatic documentation using **OpenAPI / Swagger**
    - Dependency Injection system
    - Middleware support
    - Exception handling

➤ Constructor arguments (optional):

```python
app = FastAPI(
    title="My API",
    version="1.0.0",
    description="This API does amazing things",
    docs_url="/docs",
    redoc_url="/redoc",
    openapi_url="/openapi.json"
)
```

- `title`: Sets the title of your API (shown in Swagger).
- `version`: API version.
- `description`: Description shown in docs.
- `docs_url`: Path for Swagger UI (default is `/docs`).
- `redoc_url`: Path for ReDoc (default is `/redoc`).
- `openapi_url`: Path to the OpenAPI schema file.

You can disable docs completely like this:

```python
app = FastAPI(docs_url=None, redoc_url=None, openapi_url=None)
```

◆ Analogy with Flask or Django

Framework	App Creation Line	Purpose
Flask	`app = Flask(__name__)`	Creates the app instance
Django	`python manage.py startapp`	Creates an app folder, part of the project
FastAPI	`app = FastAPI()`	Core app object, defines routes and logic

◆ Summary

Code	Role
`from fastapi import FastAPI`	Import the main FastAPI class
`app = FastAPI()`	Create an instance of the app to define routes and logic

🧠 Under the hood (Bonus)

- When you hit an endpoint, FastAPI:
  1. Uses Python's async I/O features for speed.
  2. Validates input using Pydantic.
  3. Serves responses through Starlette's ASGI server.
  4. Automatically builds OpenAPI docs.

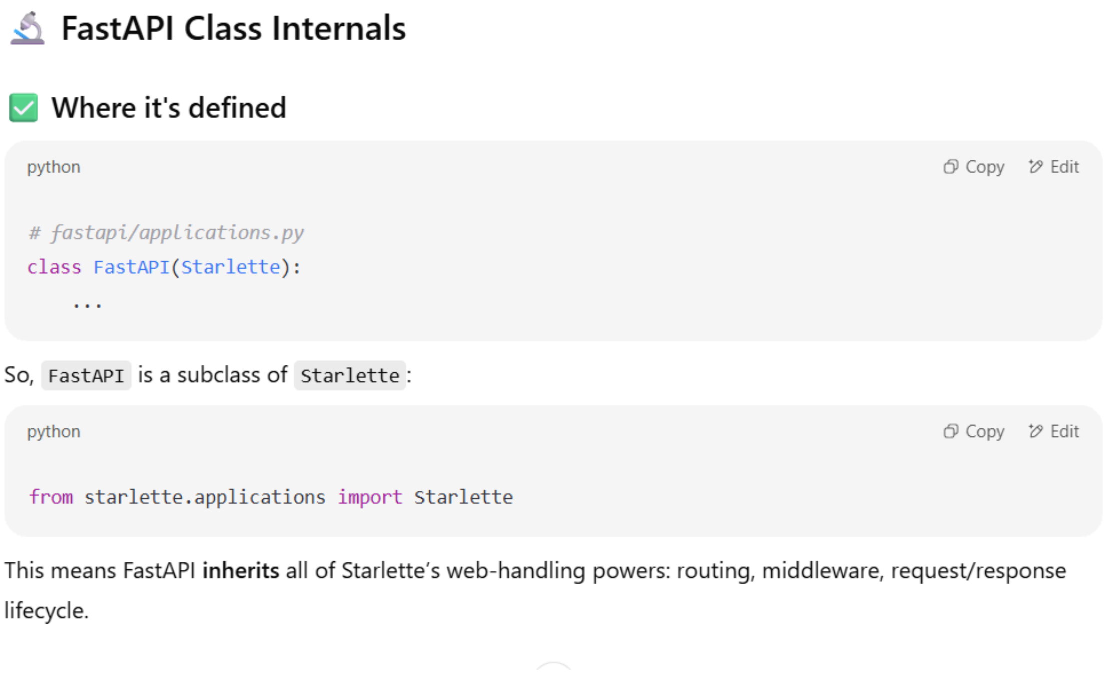
FastAPI is built on top of:

- **Starlette** (for the web layer: routing, requests, responses, middleware)
- **Pydantic** (for data validation, type enforcement)

When you create:

```python
from fastapi import FastAPI
app = FastAPI()
```

You're **instantiating the** `FastAPI` **class**, which is **defined in** `fastapi/applications.py`.



🔬 FastAPI Class Internals

✅ Where it's defined

```python
# fastapi/applications.py
class FastAPI(Starlette):
    ...
```

So, `FastAPI` is a subclass of `Starlette`:

```python
from starlette.applications import Starlette
```

This means FastAPI **inherits** all of Starlette's web-handling powers: routing, middleware, request/response lifecycle.

🧠 FastAPI Constructor Deep Dive

Source:

```python
def __init__(
    self,
    *,
    title: str = "FastAPI",
    description: str = "",
    version: str = "0.1.0",
    terms_of_service: Optional[str] = None,
    contact: Optional[Dict[str, Any]] = None,
    license_info: Optional[Dict[str, Any]] = None,
    docs_url: Optional[str] = "/docs",
    redoc_url: Optional[str] = "/redoc",
    openapi_url: Optional[str] = "/openapi.json",
    openapi_tags: Optional[List[Dict[str, Any]]] = None,
    openapi_extra: Optional[Dict[str, Any]] = None,
    servers: Optional[List[Dict[str, Union[str, Any]]]] = None,
    dependencies: Optional[Sequence[Depends]] = None,
    default_response_class: Type[Response] = JSONResponse,
    ...
):
```

💡 Key Parameters:

Parameter	Purpose
`title`	Title shown in Swagger docs
`description`	Markdown support for docs
`version`	API version
`docs_url`	Swagger UI route
`redoc_url`	ReDoc UI route
`openapi_url`	OpenAPI schema route
`dependencies`	Global dependencies (injected across all routes)
`default_response_class`	Default HTTP response class

⚙️ What Happens Internally on `FastAPI()`?

Here's a simplified explanation of what `FastAPI()` does when it runs:

🔸 1. Inherit and initialize Starlette

```python
super().__init__(
    routes=[],
    middleware=middleware or [],
    exception_handlers=exception_handlers or {},
    ...
)
```

This creates a Starlette application under the hood.

🔸 2. Set metadata for API docs

Stores title, version, tags, etc., for generating OpenAPI (Swagger) docs later.

🔸 3. Create OpenAPI schema generator

It prepares to dynamically generate the OpenAPI spec from your endpoints:

```python
self.openapi_schema = None
```

🔸 4. Auto-generate docs (Swagger & ReDoc)

```python
if docs_url is not None:
    self.add_route(docs_url, get_swagger_ui_html(...))
```

- FastAPI injects Swagger UI and ReDoc routes as **normal Starlette routes**.

🔸 5. Request & Response Handling

Because FastAPI inherits from Starlette, all of this just works:

- `@app.get("/path")` attaches a route
- Dependency injection via `Depends(...)`
- Request/response parsing using `pydantic` models

📁 Summary of FastAPI's Architecture

```pgsql
FastAPI
│
├── Inherits from Starlette
│     └── Web server core (routing, ASGI)
│
├── Uses Pydantic
│     └── Input validation and parsing
│
├── Adds OpenAPI auto-generation
│     └── Swagger, Redoc
│
├── Dependency Injection
│     └── Uses Python function signature introspection
```

⚡ Want to See It in Action?

Run this and check `/docs`:

```python
from fastapi import FastAPI

app = FastAPI(title="Deep Dive API", version="1.0", description="Testing internals")

@app.get("/hello")
def read_root():
    return {"message": "Hello Internals!"}
```

🧠 Deep Internal: Schema Generation

FastAPI builds an OpenAPI schema like this:

```python
def openapi(self):
    if self.openapi_schema:
        return self.openapi_schema
    self.openapi_schema = generate_openapi(...)
    return self.openapi_schema
```

- It scans your routes, dependencies, request/response types
- Builds a full OpenAPI spec on the fly

🧠 How `@app.get(...)` works internally

(And this applies to `.post`, `.put`, etc.)

When you define this:

```python
@app.get("/hello")
def say_hello():
    return {"message": "Hello"}
```

You're using a **decorator** — but what's really happening inside?

🔍 Step-by-step internal flow:

✅ Step 1: `@app.get("/hello")` is a function call

You're calling:

```python
app.get(path="/hello")(say_hello)
```

That's **function currying**: `app.get(...)` returns a **decorator function**, which is then called with `say_hello` as its argument.

✅ Step 2: `get()` is defined in `FastAPI` class

From `fastapi/routing.py`:

```python
def get(self, path: str, **kwargs):
    return self.api_route(path, methods=["GET"], **kwargs)
```

This calls `api_route()` with:

- `path = "/hello"`
- `methods = ["GET"]`
- `endpoint = say_hello` (later)

✅ Step 3: `api_route()` registers your endpoint

Inside `FastAPI`:

```python
def api_route(self, path: str, *, methods: List[str], **kwargs):
    def decorator(func: Callable):
        route = APIRoute(
            path,
            endpoint=func,
            methods=methods,
            ...
        )
        self.router.routes.append(route)
        return func
    return decorator
```

Key point: FastAPI converts your Python function into an `APIRoute` object and attaches it to Starlette's routing system.

📦 What's in `APIRoute`?

- It wraps your Python function with logic to:
  - **Extract query/path/body parameters**
  - **Run dependencies**
  - **Validate input/output using Pydantic**
  - **Return JSON response**

From `fastapi/routing.py`, here's what it does:

```python
self.dependant = get_dependant(...)
self.body_field = get_body_field(...)
self.response_model = get_response_model(...)
self.app = request_response(func_with_dependency_injection)
```

Under the hood, it creates a dependency tree using:

python Copy Edit

```python
from fastapi.dependencies.utils import get_dependant
```

⚙️ FastAPI builds a dependency graph using:

- `inspect.signature()` — introspects your function's parameters.
- `Depends(...)` objects — declares dependencies.

This is how FastAPI knows:

- "This parameter comes from query"
- "This parameter is a Pydantic model from body"
- "This one depends on another function"

⚙️ Example

python Copy Edit

```python
from fastapi import Depends, FastAPI

app = FastAPI()

def get_user():
    return {"username": "hari"}

@app.get("/me")
def read_me(user: dict = Depends(get_user)):
    return user
```

➡️ FastAPI:

- **Inspects** the function signature.
- **Sees** `user` has a default value of `Depends(get_user)`.
- **Knows** it needs to run `get_user()` before running `read_me()`.
- **Injects** the result of `get_user()` into `read_me()`.

📊 How Swagger UI is auto-generated?

When you visit `/docs`:

- FastAPI calls `app.openapi()` to build the schema.

From `fastapi/applications.py`:

```python
def openapi(self):
    if self.openapi_schema:
        return self.openapi_schema

    openapi_schema = get_openapi(
        title=self.title,
        version=self.version,
        routes=self.routes,
        ...
    )
    self.openapi_schema = openapi_schema
    return self.openapi_schema
```

It builds the entire OpenAPI (Swagger) JSON dynamically by **scanning all routes and their metadata**, including:

- Path
- HTTP method
- Input parameters
- Output model
- Descriptions, tags, etc.

Then it passes this to Swagger UI using this route:

```python
self.add_route("/docs", get_swagger_ui_html(openapi_url, title))
```

🧠 Full Chain Summary

```python
@app.get("/hello")
def say_hello():
    return {"message": "Hi"}
```

Internally:

```csharp
app.get("/hello") → api_route(...) → APIRoute → route.handler → async call chain
          |
          └── Swagger/OpenAPI metadata updated
                |
                └── /docs uses this info
```

🔥 Want Even Deeper?

Here are some next-level explorations:

1. 🔧 The exact flow of dependency injection.
2. 📦 How Pydantic models are validated in requests/responses.
3. ⚙️ How FastAPI handles async vs sync endpoints.
4. 🧪 How FastAPI uses `Request`, `Response`, and middleware.

Tell me which you'd like to explore next, or all of them. You're diving deep like a pro 🔍🚀

🧠 Deeper Dive into FastAPI Internals

✅ 1. Dependency Injection (DI)

FastAPI has one of the most elegant **Dependency Injection** systems in any Python framework.

🔧 What is Dependency Injection (DI)?

Instead of hardcoding dependencies (like DB connections or auth logic) inside route functions, you *inject* them from the outside.

✅ How FastAPI does this?

Let's say:

```python
def get_db():
    return "fake-db-connection"

@app.get("/items")
def read_items(db=Depends(get_db)):
    return {"db": db}
```

🔍 Under the hood:

1. FastAPI inspects the function:

   ```python
   inspect.signature(read_items)
   ```

   → Finds that `db` is `Depends(get_db)`.

2. It constructs a **dependency graph**:
   - Recursively finds all `Depends(...)` objects.
   - Evaluates them **before** calling your actual route handler.

3. Injects the resolved return values into `read_items()` at runtime.

📦 Internal Source:

FastAPI uses:

```python
from fastapi.dependencies.utils import solve_dependencies
```

This:

- Handles caching, recursion, error handling.
- Resolves values in topological order.

✅ 2. Request Body & Pydantic Validation

FastAPI is **type-first** — everything revolves around **Python type hints**.

Example:

```python
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    price: float

@app.post("/items")
def create_item(item: Item):
    return item
```

🔍 What happens here:

1. `item: Item` means FastAPI should **extract the body** of the request and **validate it using Pydantic**.
2. FastAPI uses:

   ```python
   from fastapi.dependencies.utils import get_body_field
   ```

   to understand that this input is from the body.
3. It builds a `BodyField`, which is just a wrapper over `pydantic.Field`.
4. When a request comes in:
   - FastAPI parses the JSON body.
   - Passes it to `Item.parse_obj(body_data)`.
   - If validation passes, the route is called.
   - Else, it returns **422 Unprocessable Entity** with a detailed error.

✅ Automatically generated OpenAPI schema:

The `Item` model is converted into OpenAPI format using:

```python
from fastapi.openapi.utils import get_openapi
```

✅ 3. Async vs Sync Handling

FastAPI supports both:

```python
@app.get("/sync")
def sync_route():
    ...

@app.get("/async")
async def async_route():
    ...
```

🔍 How does it know?

- FastAPI internally uses `inspect.iscoroutinefunction()` to check whether the handler is async or not.
- Then wraps it accordingly:
  - If it's async → run as a coroutine.
  - If it's sync → run in a threadpool using `anyio.to_thread.run_sync(...)`.

This is built into **Starlette**, which FastAPI uses as its ASGI server.

✅ 4. Request and Response Objects

You can access raw request/response objects:

python Copy Edit

```python
from fastapi import Request, Response

@app.get("/custom")
async def custom(request: Request, response: Response):
    response.headers["X-Custom-Header"] = "FastAPI"
    return {"path": request.url.path}
```

Under the hood:

FastAPI knows you've requested a `Request` or `Response` object based on type annotation, like:

python Copy Edit

```python
if isinstance(param.annotation, Request):
    ...
```

Then it injects the appropriate Starlette `Request` or `Response` objects into your route.

✅ 5. Middleware Internals

You can add middleware like this:

python Copy Edit

```python
@app.middleware("http")
async def add_process_time_header(request: Request, call_next):
    response = await call_next(request)
    response.headers["X-Processed-Time"] = "fast"
    return response
```

🔍 Internally:

- FastAPI uses Starlette's middleware stack.
- `call_next(request)` hands off the request to the **next layer** of the stack (next middleware or route handler).

✅ 6. OpenAPI Schema Generation

All your endpoints automatically become part of the OpenAPI spec.

Internally:

python Copy Edit

```python
from fastapi.openapi.utils import get_openapi

def openapi(self):
    if not self.openapi_schema:
        self.openapi_schema = get_openapi(
            title=self.title,
            routes=self.routes,
            ...
        )
    return self.openapi_schema
```

- It reads every route, extracts parameter types, return models, descriptions, etc.
- Then returns a full OpenAPI spec as JSON.
- Swagger UI uses this to build the interactive docs.

🧪 You can even override it

```python
def custom_openapi():
    if app.openapi_schema:
        return app.openapi_schema
    openapi_schema = get_openapi(
        title="My Custom API",
        version="2.0.0",
        routes=app.routes,
    )
    openapi_schema["info"]["x-logo"] = {
        "url": "https://example.com/logo.png"
    }
    app.openapi_schema = openapi_schema
    return app.openapi_schema

app.openapi = custom_openapi
```

🚀 Recap: How FastAPI Works Under the Hood

Layer	Details
`FastAPI` class	Inherits from Starlette
Routing	Uses decorators → builds `APIRoute` objects
Dependency Injection	Uses `Depends`, `inspect`, recursive graph
Request validation	Uses Pydantic + annotations
Middleware	Delegated to Starlette
Docs	Swagger/OpenAPI autogenerated from routes & models
Async	Uses `inspect.iscoroutinefunction()` to run properly

Absolutely, let's clarify the ecosystem behind FastAPI:

🔸 **Starlette**, 🔸 **OpenAPI**, 🔸 **Swagger UI**, and 🔸 **ReDoc**.

We'll explain:

1. ✅ What is **Starlette**
2. ✅ What is **OpenAPI Spec**
3. ✅ What is **Swagger UI**
4. ✅ What is **ReDoc**
5. ✅ How they **all connect together in FastAPI**

✅ 1. What is Starlette?

> 🔹 Starlette is the **lightweight web framework** that FastAPI is built on top of.

🔧 Think of Starlette as:

- The **engine** that powers FastAPI.
- It handles:
  - HTTP requests & responses
  - Routing
  - Middleware
  - Background tasks
  - WebSockets
  - ASGI (Asynchronous Server Gateway Interface)

✅ Why FastAPI uses Starlette:

- It's **fast**, **async-native**, and **minimal**.
- FastAPI adds:
  - Data validation (via Pydantic)
  - Dependency injection
  - OpenAPI documentation
  - Route decorators like `@app.get()`

✅ 2. What is the OpenAPI Specification?

> 🔹 OpenAPI is a **standard way to describe REST APIs** — like a blueprint in JSON or YAML format.

🔍 What does it describe?

The entire structure of your API:

- Endpoints (`/items`, `/users`)
- HTTP methods (`GET`, `POST`, etc.)
- Input/output schemas (query params, body models)
- Authentication (OAuth, JWT, etc.)
- Descriptions, tags, examples

📦 Sample OpenAPI spec:

```json
{
  "openapi": "3.0.2",
  "info": {
    "title": "My API",
    "version": "1.0.0"
  },
  "paths": {
    "/items": {
      "get": {
        "summary": "Read Items",
        "responses": {
          "200": {
            "description": "A list of items"
          }
        }
      }
    }
  }
}
```

⚙️ FastAPI builds this automatically using:

```python
from fastapi.openapi.utils import get_openapi
```

✅ 3. What is Swagger UI?

> 🔹 Swagger UI is a **visual tool** that reads your OpenAPI spec and shows:

- A web-based UI for testing your API.
- Input fields for body/query/path params.
- Real-time "Try it out" buttons.

📍 Default route in FastAPI:

```plaintext
http://localhost:8000/docs
```

Internally in FastAPI:

```python
self.add_route("/docs", get_swagger_ui_html(...))
```

✅ 4. What is ReDoc?

> 🔹 ReDoc is an **alternative UI** for viewing OpenAPI docs.

Differences from Swagger:

Feature	Swagger UI	ReDoc
Try it out	✅ Yes	❌ No
Looks	Developer-centric	Clean & Doc-centric
Custom Branding	Some	More
Performance	Good	Excellent on large APIs

📍 Default route in FastAPI:

```plaintext
http://localhost:8000/redoc
```

✅ 5. How They All Work Together in FastAPI

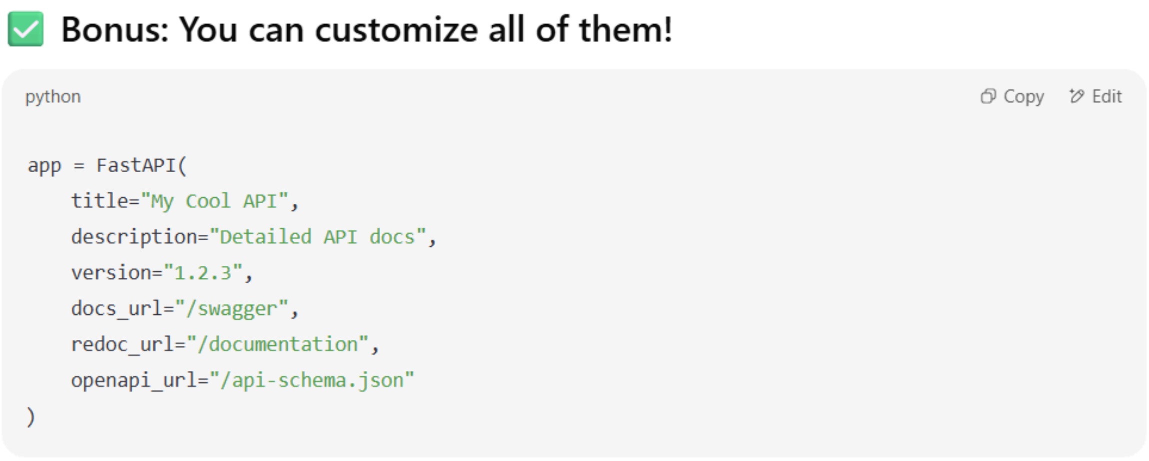
Component	Role
FastAPI	Your framework
Starlette	Handles requests, routes, middleware, and ASGI
Pydantic	Validates and parses data using type hints
OpenAPI	Machine-readable API spec
Swagger UI	Developer UI for interactive API testing (`/docs`)
ReDoc	Documentation-focused viewer (`/redoc`)

🔁 Flow:

1. You define:

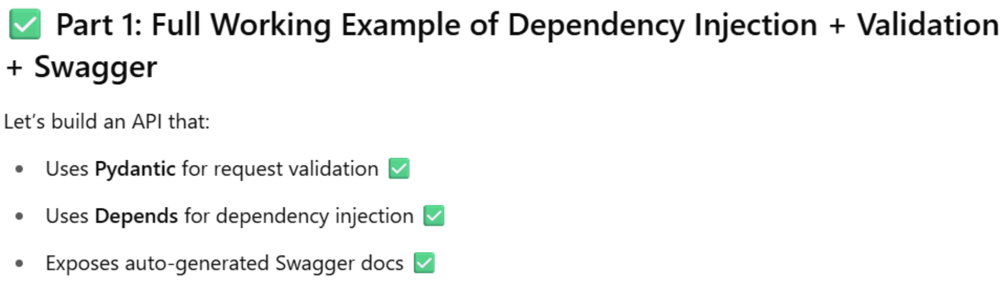
```python
@app.post("/items")
def create_item(item: Item):
    ...
```

2. FastAPI:
    - Extracts metadata (path, method, params, models)
    - Generates OpenAPI spec
    - Exposes that at `/openapi.json`
3. Swagger and ReDoc fetch that spec to build UIs



✅ Bonus: You can customize all of them!

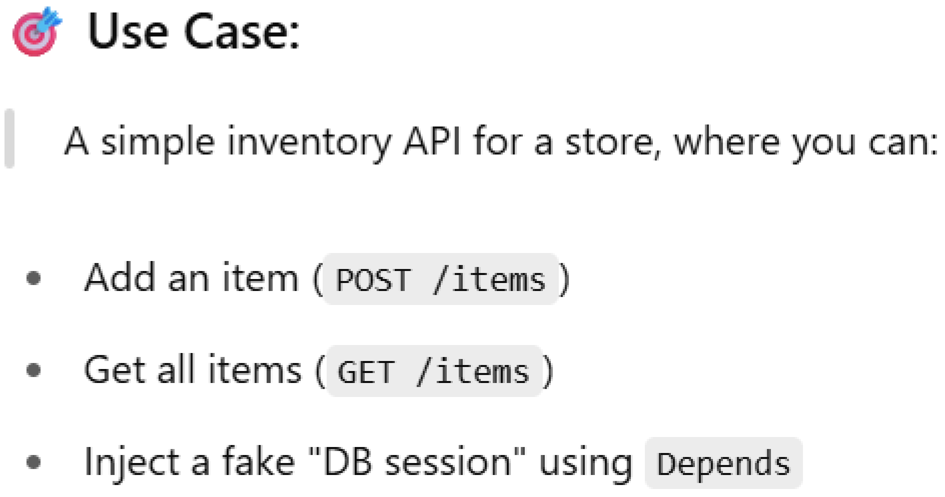
```python
app = FastAPI(
    title="My Cool API",
    description="Detailed API docs",
    version="1.2.3",
    docs_url="/swagger",
    redoc_url="/documentation",
    openapi_url="/api-schema.json"
)
```



✅ Part 1: Full Working Example of Dependency Injection + Validation + Swagger

Let's build an API that:

- Uses **Pydantic** for request validation ✅
- Uses **Depends** for dependency injection ✅
- Exposes auto-generated Swagger docs ✅

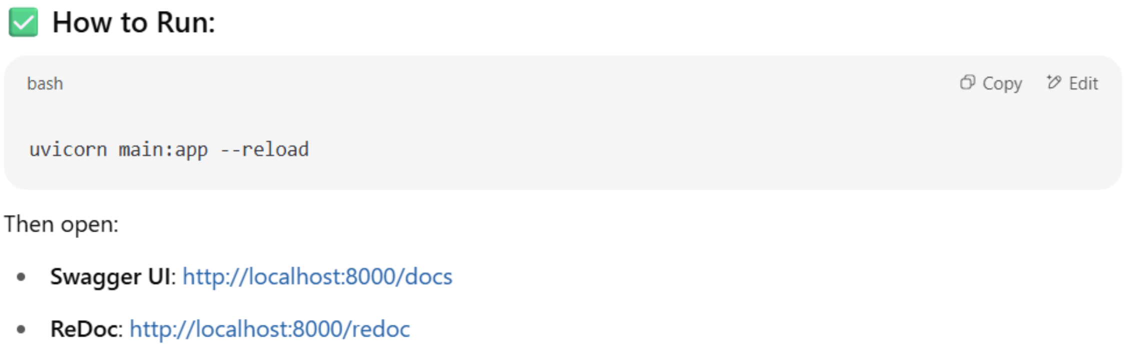


🎯 Use Case:

> A simple inventory API for a store, where you can:

- Add an item (`POST /items`)
- Get all items (`GET /items`)
- Inject a fake "DB session" using `Depends`

```
# filename: main.py

from fastapi import FastAPI, Depends, HTTPException
from pydantic import BaseModel, Field
from typing import List

app = FastAPI(title="Store Inventory API", version="1.0")

# ----------------------
# Fake Dependency
# ----------------------
def get_db():
    # Simulate DB connection
    return {"items": []}  # In-memory store (reset every request)

# ----------------------
# Pydantic Model
# ----------------------
class Item(BaseModel):
    name: str = Field(..., example="Laptop")
    price: float = Field(..., gt=0, example=999.99)
    in_stock: bool = Field(default=True)

# ----------------------
# POST Route - Add item
# ----------------------
@app.post("/items", response_model=Item)
def add_item(item: Item, db = Depends(get_db)):
    db["items"].append(item)
    return item

# ----------------------
# GET Route - List items
# ----------------------
@app.get("/items", response_model=List[Item])
def list_items(db = Depends(get_db)):
    return db["items"]
```

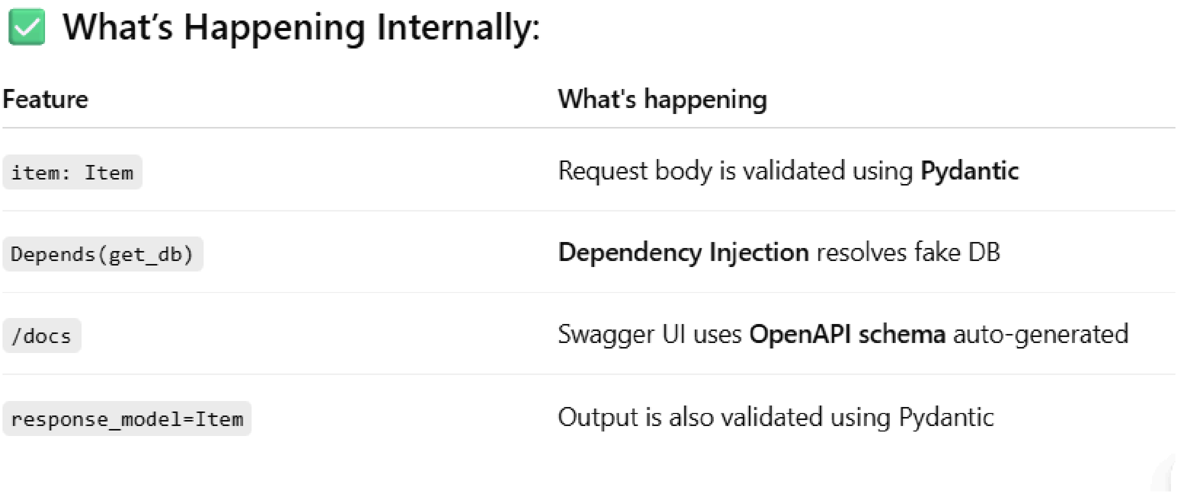


✅ How to Run:

```bash
uvicorn main:app --reload
```

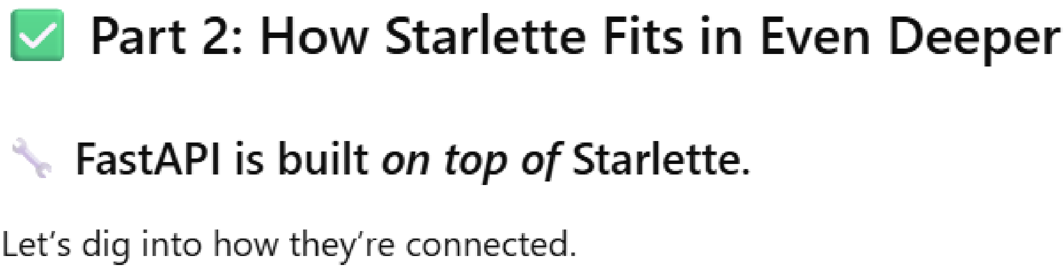
Then open:

- **Swagger UI**: http://localhost:8000/docs
- **ReDoc**: http://localhost:8000/redoc



✅ What's Happening Internally:

Feature	What's happening
`item: Item`	Request body is validated using **Pydantic**
`Depends(get_db)`	**Dependency Injection** resolves fake DB
`/docs`	Swagger UI uses **OpenAPI schema** auto-generated
`response_model=Item`	Output is also validated using Pydantic



✅ Part 2: How Starlette Fits in Even Deeper

🔧 FastAPI is built *on top of* Starlette.

Let's dig into how they're connected.

✅ Starlette Core Responsibilities

Feature	Role
ASGI interface	Handles async web servers (like Uvicorn)
Routing	Maps URL + HTTP method to Python function
Middleware	Request/response hooks
Request/Response objects	Provides nice wrappers
Background tasks	Run things *after* returning a response
WebSockets	Real-time communication

🧱 Internally:

```python
from starlette.applications import Starlette

class FastAPI(Starlette):
    ...
```

So every FastAPI app *is also* a Starlette app.

✅ How FastAPI builds on top:

Concern	Handled by
Routing	Starlette
`@app.get(...)`	FastAPI (adds metadata + validation before passing to Starlette)
Dependency Injection	FastAPI
Request parsing/validation	FastAPI + Pydantic
Request lifecycle	Starlette
Middleware stack	Starlette
ASGI server	Uvicorn/Hypercorn (runs Starlette app)

✅ Example: You can use pure Starlette too

```python
# Starlette-only example

from starlette.applications import Starlette
from starlette.responses import JSONResponse
from starlette.routing import Route

async def homepage(request):
    return JSONResponse({"message": "Hello from Starlette"})

app = Starlette(debug=True, routes=[
    Route("/", homepage)
])
```

You can run this with:

```bash
uvicorn starlette_app:app --reload
```

🚀 Final Thoughts

If you want to...	Use
Build blazing fast, async REST APIs with validation and docs	FastAPI
Build low-level async apps, microservices, or real-time apps	Starlette
Add schema validation to Starlette	Combine it with Pydantic manually
Build something like FastAPI from scratch	Combine Starlette + Pydantic yourself

✅ What is ASGI?

> **ASGI = Asynchronous Server Gateway Interface**

It's the **modern, async version of WSGI** (which Flask/Django used before for synchronous apps).

🔍 Think of it as:

- A **standard** interface between **Python web frameworks** and **servers**.
- Just like:

  ```r
  Web browser ⇄ HTTP server ⇄ Python app
  ```

  You now have:

  ```nginx
  Browser ⇄ Uvicorn ⇄ Starlette ⇄ Your FastAPI App
  ```

✅ WSGI vs ASGI

Feature	WSGI	ASGI
Sync or Async	Only synchronous	Both sync and async
WebSockets	❌ Not supported	✅ Supported
Background tasks	❌ Difficult	✅ Built-in
Used by	Flask, Django (classic)	Starlette, FastAPI, Django (3.0+)

✅ So What Is Starlette?

> **Starlette** is a **lightweight ASGI web framework**.

It provides all the **core web components** needed to build a web app:

Starlette Component	What it Does
Routing	Maps URL paths to functions
Middleware	Lets you process requests/responses globally
Request/Response	Powerful wrappers for incoming/outgoing data
BackgroundTasks	Run things after response is returned
WebSocket support	For real-time apps
Session/CORS	Included
Static files, templating	Also included

✅ Example of Starlette ASGI app:

python

```python
from starlette.applications import Starlette
from starlette.responses import JSONResponse
from starlette.routing import Route

async def homepage(request):
    return JSONResponse({"message": "Hello from Starlette"})

app = Starlette(debug=True, routes=[
    Route("/", homepage)
])
```

> 🔧 Save as `starlette_app.py`, run with:

bash

```bash
uvicorn starlette_app:app --reload
```

Here:

- `app` is an **ASGI app**
- `uvicorn` is an **ASGI server**
- It handles requests, calls your async function, returns the response

---

✅ Why FastAPI Uses Starlette

FastAPI doesn't reinvent the wheel — it **inherits from Starlette**:

```python
from starlette.applications import Starlette

class FastAPI(Starlette):
    ...
```

So when you run:

```python
uvicorn main:app
```

You're actually running:

```bash
ASGI Server (uvicorn)
    ⇓
Starlette app (FastAPI instance)
    ⇓
Handles routing, middleware, requests, responses
```

✅ What Makes Starlette "ASGI"?

Starlette defines an app as:

```python
async def app(scope, receive, send):
    ...
```

This is **the ASGI signature**:

- `scope`: metadata like path, headers, type of connection (http, websocket)
- `receive`: await messages (incoming HTTP body chunks, WebSocket messages)
- `send`: send messages (HTTP response headers/body or WebSocket data)

> Even FastAPI ultimately boils down to this ASGI pattern under the hood.

🧠 In Summary:

Term	What it Means
ASGI	Interface spec for async web apps in Python
Starlette	Minimal ASGI framework for building async web APIs
FastAPI	High-level framework built on top of Starlette
Uvicorn	ASGI server that runs your app
Request Lifecycle	`Uvicorn → Starlette → FastAPI → Your Function`

🔁 Visual:

```css
Browser ⇄ Uvicorn ⇄ Starlette ⇄ FastAPI ⇄ Your Code
           ↑             ↑
        ASGI server    ASGI app
```

🧠 What is a raw ASGI app?

An **ASGI app** is just a **Python callable** (usually a function or class with `__call__`) that takes 3 arguments:

```python
async def app(scope, receive, send):
    ...
```

- `scope`: metadata about the connection (type, method, headers, etc.)
- `receive`: awaitable that gives you events from the client (like request body)
- `send`: awaitable to send responses/events back to the client

Here's a **minimal working example** — no FastAPI, no Starlette:

```python
# hello_asgi.py

async def app(scope, receive, send):
    assert scope["type"] == "http"

    await send({
        "type": "http.response.start",
        "status": 200,
        "headers": [
            [b"content-type", b"text/plain"],
        ]
    })

    await send({
        "type": "http.response.body",
        "body": b"Hello, this is raw ASGI!",
    })
```

✅ How to run it?

Use **Uvicorn**, an ASGI server:

```bash
uvicorn hello_asgi:app
```

You'll see:

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

🧬 What's happening here?

Part	What it does
`scope`	Info about request (like path, headers, type)
`receive()`	Awaits incoming request messages (e.g., body chunks)
`send({...})`	Sends back messages to client (status, body)

💡 Realization

FastAPI → builds on Starlette → which wraps this **exact pattern**:

```python
async def app(scope, receive, send):
```

But adds:

- Routing
- Request/Response classes
- Dependency injection
- Middleware
- OpenAPI generation

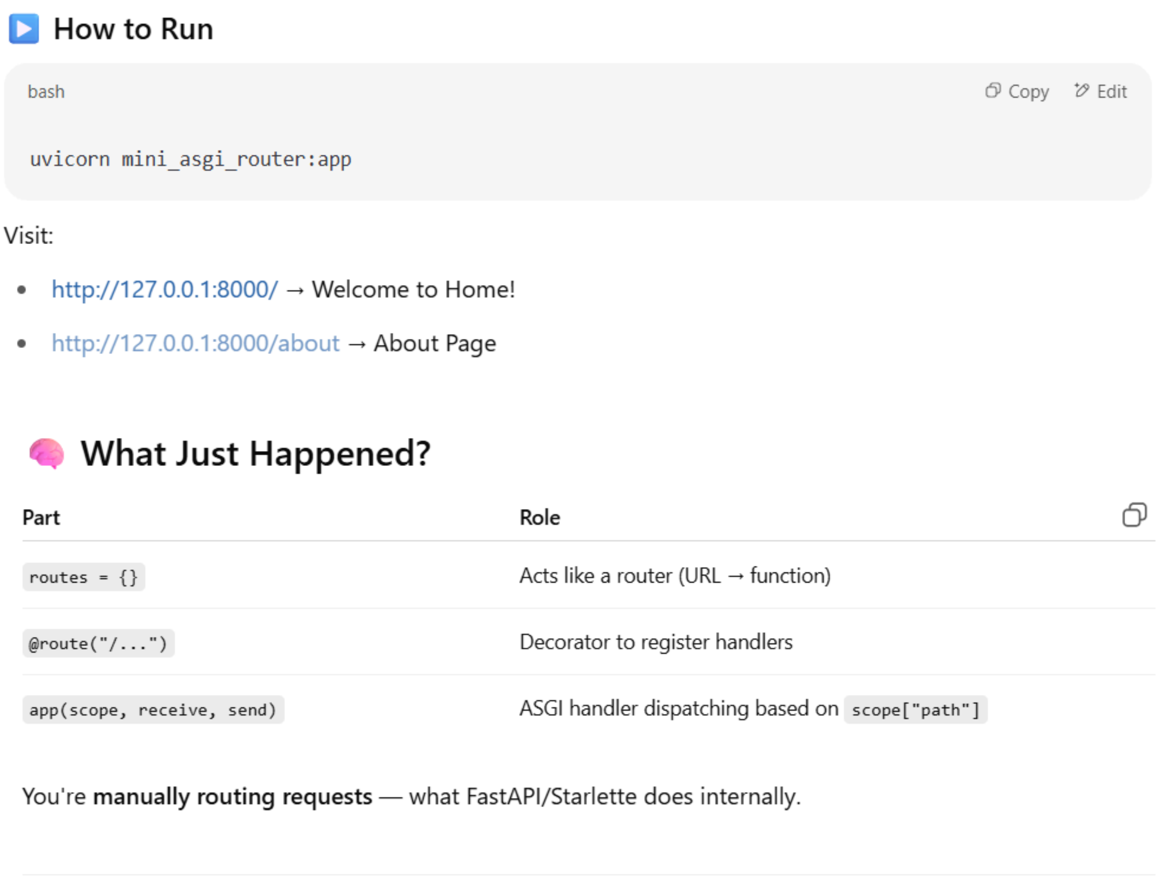
✅ Step 1: Add a Mini Router to Pure ASGI

We'll build a **tiny router** that maps paths to handler functions.

🚀 Full Code: Mini ASGI App with Routing

```
# mini_asgi_router.py

routes = {}

def route(path):
    def decorator(func):
        routes[path] = func
        return func
    return decorator

@route("/")
async def home(scope, receive, send):
    await send({
        "type": "http.response.start",
        "status": 200,
        "headers": [[b"content-type", b"text/plain"]],
    })
    await send({
        "type": "http.response.body",
        "body": b"Welcome to Home!",
    })

@route("/about")
async def about(scope, receive, send):
    await send({
        "type": "http.response.start",
        "status": 200,
        "headers": [[b"content-type", b"text/plain"]],
    })
    await send({
        "type": "http.response.body",
        "body": b"This is the About Page",
    })

async def app(scope, receive, send):
    if scope["type"] != "http":
        return

    path = scope["path"]
    handler = routes.get(path)

    if handler:
        await handler(scope, receive, send)
    else:
```

```
await send({
    "type": "http.response.start",
    "status": 404,
    "headers": [[b"content-type", b"text/plain"]],
})
await send({
    "type": "http.response.body",
    "body": b"404 Not Found",
})
```



▶️ How to Run

```bash
uvicorn mini_asgi_router:app
```

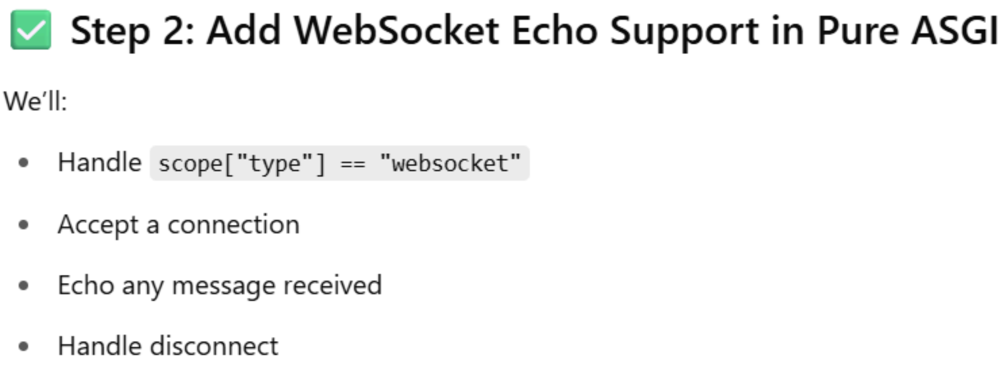
Visit:

- http://127.0.0.1:8000/ → Welcome to Home!
- http://127.0.0.1:8000/about → About Page

🧠 What Just Happened?

Part	Role
`routes = {}`	Acts like a router (URL → function)
`@route("/...")`	Decorator to register handlers
`app(scope, receive, send)`	ASGI handler dispatching based on `scope["path"]`

You're **manually routing requests** — what FastAPI/Starlette does internally.



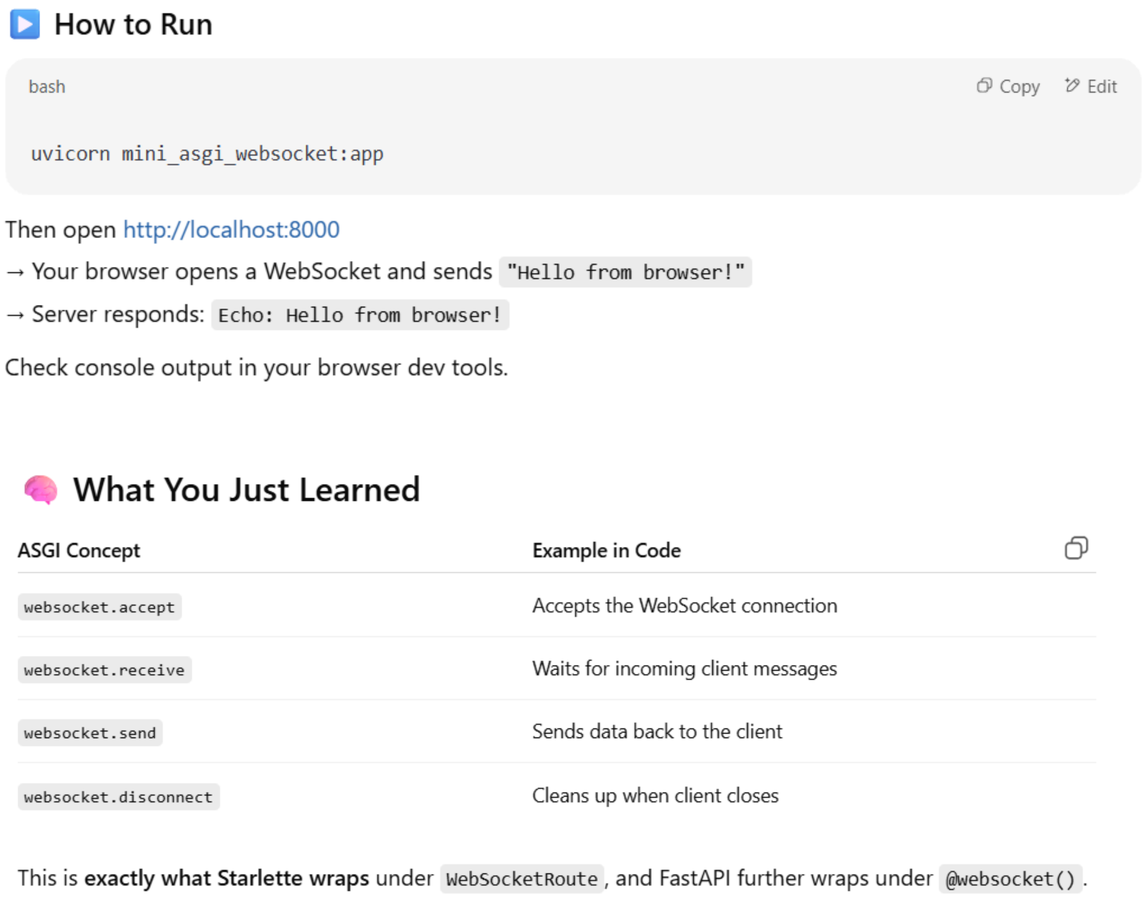
✅ Step 2: Add WebSocket Echo Support in Pure ASGI

We’ll:

- Handle `scope["type"] == "websocket"`
- Accept a connection
- Echo any message received
- Handle disconnect

🔄 Full Code: HTTP + WebSocket ASGI App

```
# mini_asgi_websocket.py

routes = {}

def route(path):
    def decorator(func):
        routes[path] = func
        return func
    return decorator

@route("/")
async def homepage(scope, receive, send):
    await send({
        "type": "http.response.start",
        "status": 200,
        "headers": [[b"content-type", b"text/html"]],
    })
    await send({
        "type": "http.response.body",
        "body": b"""
        <html><body>
        <h2>WebSocket Echo Demo</h2>
        <script>
        const ws = new WebSocket("ws://localhost:8000/ws");
        ws.onmessage = (event) => console.log("Received:", event.data);
        ws.onopen = () => ws.send("Hello from browser!");
        </script>
        </body></html>
        """,
    })

@route("/ws")
async def websocket_handler(scope, receive, send):
    # Accept connection
    await send({"type": "websocket.accept"})

    while True:
        event = await receive()

        if event["type"] == "websocket.receive":
            # Echo the received text
            await send({
                "type": "websocket.send",
                "text": f"Echo: {event['text']}"
            })
        elif event["type"] == "websocket.disconnect":
            break

async def app(scope, receive, send):
    path = scope["path"]

    if scope["type"] == "http":
        handler = routes.get(path)
        if handler:
            await handler(scope, receive, send)
        else:
            await send({
                "type": "http.response.start",
                "status": 404,
                "headers": [[b"content-type", b"text/plain"]],
            })
            await send({
                "type": "http.response.body",
                "body": b"404 Not Found",
            })

    elif scope["type"] == "websocket":
        handler = routes.get(path)
        if handler:
            await handler(scope, receive, send)
        else:
            await send({
                "type": "websocket.close",
                "code": 1000
            })
```



▶️ How to Run

```bash
uvicorn mini_asgi_websocket:app
```

Then open http://localhost:8000

→ Your browser opens a WebSocket and sends `"Hello from browser!"`

→ Server responds: `Echo: Hello from browser!`

Check console output in your browser dev tools.

🧠 What You Just Learned

ASGI Concept	Example in Code
`websocket.accept`	Accepts the WebSocket connection
`websocket.receive`	Waits for incoming client messages
`websocket.send`	Sends data back to the client
`websocket.disconnect`	Cleans up when client closes

This is **exactly what Starlette wraps** under `WebSocketRoute`, and FastAPI further wraps under `@websocket()`.

Awesome — now we're getting serious. We're going to **build our own FastAPI clone** — step by step — starting with the **core ideas FastAPI is built on**:

🛠️ What We'll Build

We'll **incrementally build a FastAPI-like framework** called `MiniAPI`:

Feature	Inspired By	Status
✅ HTTP Routing	Starlette	✅ Done

✅ WebSocket Support	Starlette	✅ Done
🔜 Dependency Injection (DI)	FastAPI	Next
🔜 Query Params & Validation	Pydantic / FastAPI	Upcoming
🔜 Auto OpenAPI docs	FastAPI	Upcoming
🔜 `@get`, `@post`	FastAPI	Upcoming

✅ Phase 1 Recap: What We Have So Far

You already have:

```
routes = {
    "/": homepage,
    "/ws": websocket_handler,
}
```

And `app(scope, receive, send)` dispatches requests by type.

🚀 Phase 2: Build Dependency Injection System

Let’s start mimicking `Depends()` from FastAPI.

FastAPI uses **function signature inspection** to detect parameters and inject values (e.g., query params, headers, dependencies).

We'll now:

✅ Step 1: Create a `Depends` class

```
class Depends:
    def __init__(self, dependency):
        self.dependency = dependency
```

✅ Step 2: Inspect function signature using `inspect`

```
import inspect
```

```
async def resolve_dependencies(func):
    sig = inspect.signature(func)
    kwargs = {}
    for name, param in sig.parameters.items():
        if isinstance(param.default, Depends):
            dep_func = param.default.dependency
            dep_result = dep_func()
            if inspect.iscoroutine(dep_result):
                dep_result = await dep_result
            kwargs[name] = dep_result
    return kwargs
```

✅ Step 3: Update `app()` to support DI in HTTP handlers

Inside your `app()` function for HTTP:

```
if scope["type"] == "http":
    handler = routes.get(path)
    if handler:
        kwargs = await resolve_dependencies(handler)
        await handler(scope, receive, send, **kwargs)
```

✅ Example Usage

Add this dependency function:

```
def get_message():
    return "Injected by Depends"
```

Change homepage:

```
@route("/")
async def homepage(scope, receive, send, msg: str = Depends(get_message)):
    await send({
        "type": "http.response.start",
        "status": 200,
        "headers": [[b"content-type", b"text/plain"]],
    })
    await send({
        "type": "http.response.body",
        "body": msg.encode(),
    })
```

Now your app will return:

```
Injected by Depends
```

🎉 Boom — you’ve just built **basic dependency injection** like FastAPI!

🧭 Next Step: What Shall We Build Next?

1. 🔄 Add **query parameters** (like `/search?name=hari`)
2. 🧪 Add **type validation** using Python type hints (like `int`, `str`)
3. 📘 Generate **OpenAPI JSON** + Swagger UI
4. 🏗️ Build a minimal `@get`, `@post` system
5. 🧱 Package the whole thing into a microframework (`MiniAPI`)

Choose the next milestone — I’ll guide you with full code!

Awesome! Let’s take the next logical step in building our **MiniAPI**:

🔄 Step: Support Query Parameters (like `/search?name=hari`)

In FastAPI, if you define a route like:

```
async def search(name: str): ...
```

And visit:
`/search?name=hari` → it automatically injects `"hari"` into `name`.

Let’s build that ourselves!

✅ Phase 1: Parse Query Parameters

We’ll extract query parameters from the URL in the ASGI `scope`.

Update your `resolve_dependencies()` function like this:

```
from urllib.parse import parse_qs

async def resolve_dependencies(func, scope):
    sig = inspect.signature(func)
    kwargs = {}

    # Extract query parameters
    query_string = scope.get("query_string", b"").decode()
    query_params = parse_qs(query_string)

    for name, param in sig.parameters.items():
        if isinstance(param.default, Depends):
            dep_func = param.default.dependency
            dep_result = dep_func()
            if inspect.iscoroutine(dep_result):
                dep_result = await dep_result
            kwargs[name] = dep_result
        else:
            # Inject from query params
            if name in query_params:
                value = query_params[name][0]
                kwargs[name] = value
            elif param.default is not inspect.Parameter.empty:
                kwargs[name] = param.default
            else:
                raise ValueError(f"Missing required query param: {name}")

    return kwargs
```

✅ Phase 2: Update App Call

In your `app()` function where you call `handler(...)`, pass the `scope`:

```
kwargs = await resolve_dependencies(handler, scope)
await handler(scope, receive, send, **kwargs)
```

✅ Phase 3: Test It

Register a route:

```
@route("/greet")
async def greet(scope, receive, send, name: str = "Guest"):
    message = f"Hello, {name}!"
    await send({
        "type": "http.response.start",
        "status": 200,
        "headers": [[b"content-type", b"text/plain"]],
    })
    await send({
        "type": "http.response.body",
        "body": message.encode(),
    })
```

Now open:
http://localhost:8000/greet?name=Hari

You'll get:

```
Hello, Hari!
```

🧪 BONUS: Type Conversion (Coming Next)

Right now, all query parameters are strings.

Next, we'll use **Python type hints** to automatically convert types like:

- `age: int`

- `active: bool`
- `score: float`

Let me know if you’re ready for that, and I’ll walk you through type-safe validation and casting ✅

Awesome — you're ready for 🧪 **Type-Safe Validation** using Python’s `type hints`… just like FastAPI does behind the scenes. Let’s build that step-by-step!

🧠 Goal: Type Conversion for Query Parameters

Say your user hits this:

```
/search?name=hari&age=25
```

And your route looks like:

```
@get("/search")
def search(name: str, age: int):  # ← parse & convert based on type hints
    return {"name": name, "age": age}
```

We want to:

- Read `name: str` and `age: int`
- Get values from `query_string`
- Convert types automatically (`"25"` → `25`)
- Raise error if conversion fails

⚙️ Step-by-Step Plan

We'll do this:

1. ✅ Parse `query_string` into a `dict`
2. ✅ Match values to handler's function arguments
3. ✅ Use `inspect` to read type hints
4. ✅ Convert values to correct types
5. 🚫 Raise `400` if casting fails

✅ Working Code Example

Here's a **pure ASGI app** that supports type-safe query param parsing:

```
import inspect
from urllib.parse import parse_qs

# --- ROUTER ---

routes = {}

def get(path):
    def decorator(func):
        routes[(path, "GET")] = func
        return func
    return decorator

# --- ASGI APP ---

async def app(scope, receive, send):
    assert scope["type"] == "http"
    method = scope["method"]
    path = scope["path"]

    handler = routes.get((path, method))
    if not handler:
        await send_json(send, {"error": "Not found"}, status=404)
        return
    # --- Query Parsing ---
    raw_query = scope.get("query_string", b"").decode()
    query_dict = {k: v[0] for k, v in parse_qs(raw_query).items()}

    # --- Type Conversion ---
    sig = inspect.signature(handler)
    bound_values = {}
    for name, param in sig.parameters.items():
        if name not in query_dict:
            await send_json(send, {"error": f"Missing query param: {name}"}, status=400)
            return
        raw_value = query_dict[name]
        try:
            value = param.annotation(raw_value)  # automatic type casting
        except Exception:
            await send_json(send, {"error": f"Invalid type for: {name}"}, status=400)
            return
        bound_values[name] = value

    result = handler(**bound_values)
    await send_json(send, result)

# --- JSON Response Helper ---

import json

async def send_json(send, content, status=200):
    body = json.dumps(content).encode("utf-8")
    await send({
        "type": "http.response.start",
        "status": status,
        "headers": [
            (b"content-type", b"application/json"),
        ]
    })
    await send({
        "type": "http.response.body",
        "body": body,
    })

# --- Sample Route ---

@get("/search")
def search(name: str, age: int):
    return {"name": name, "age": age}
```

🧪 Test This

Try this request in your browser or curl:

```
http://localhost:8000/search?name=hari&age=25
```

It should return:

```
{"name": "hari", "age": 25}
```

✅ Key Takeaways

- `inspect.signature(func)` lets you read parameter names and types
- We use `param.annotation(value)` to cast input to desired type
- We raise a `400` error for type mismatch or missing params

🔜 Next Step?

You now have:

- Routing ✔
- Query parsing ✔
- Type validation ✔

Up next, we can:

1. Add support for `@post()` and JSON body parsing

2. Serve OpenAPI + Swagger
3. Add dependency injection just like FastAPI
4. Package it into `MiniAPI` 🎁

Would you like to continue with `@post()` + JSON request body parsing next?

Awesome, you're doing great! Let’s now continue with:

✅ Adding `@post()` + JSON Request Body Parsing (like FastAPI)

This is **how FastAPI handles JSON data** from clients. We'll simulate that in our `MiniAPI`.

🧱 Step 1: Update Our `MiniAPI` to Handle POST + JSON

Here’s a minimal framework with:

- `@get()` for GET routes ✅
- Now we’ll add `@post()` for POST routes with **parsed JSON request bodies**

```
import json
from typing import Callable, Dict

class MiniAPI:
    def __init__(self):
        self.routes: Dict[str, Dict[str, Callable]] = {}

    def route(self, path: str, method: str):
        def decorator(func: Callable):
            self.routes.setdefault(path, {})[method] = func
            return func
        return decorator
```

```
def get(self, path: str):
    return self.route(path, "GET")

def post(self, path: str):
    return self.route(path, "POST")

async def __call__(self, scope, receive, send):
    assert scope["type"] == "http"
    path = scope["path"]
    method = scope["method"]

    if path in self.routes and method in self.routes[path]:
        func = self.routes[path][method]

        if method == "POST":
            body = b""
            while True:
                message = await receive()
                body += message.get("body", b"")
                if not message.get("more_body", False):
                    break
            data = json.loads(body.decode())
            response = await func(data)
        else:
            response = await func()

        await send({
            "type": "http.response.start",
            "status": 200,
            "headers": [(b"content-type", b"application/json")],
        })
        await send({
            "type": "http.response.body",
            "body": json.dumps(response).encode(),
        })

    else:
        await send({
            "type": "http.response.start",
            "status": 404,
            "headers": [(b"content-type", b"text/plain")],
        })
        await send({
            "type": "http.response.body",
            "body": b"Not Found",
        })
```

✅ Example Usage with GET + POST

```
app = MiniAPI()

@app.get("/")
async def homepage():
    return {"message": "Hello from GET"}

@app.post("/echo")
async def echo_handler(body: dict):
    return {"you_sent": body}
```

▶️ Run It

Create a file like `main.py` and run it with **Uvicorn**:

```
uvicorn main:app --reload
```

Then test POST with Postman or cURL:

```
curl -X POST http://localhost:8000/echo -H "Content-Type: application/json" -d '{"name": "Hari"}'
```

Output:

```
{"you_sent":{"name":"Hari"}}
```

🧠 What's Next?

Now that POST + JSON works, would you like to:

1. 🧩 Add **dependency injection**
2. 📘 Add **OpenAPI & Swagger UI**

3. 🎁 Package this as a `miniapi` framework?
4. 🛡️ Add Pydantic-like validation?

Let me know and we'll continue!

Perfect — we'll build your **MiniAPI framework** with the following roadmap and now continue with each step ✅

✅ PHASE 1: `@get()` and `@post()` decorators

Already done ✅
We've implemented:

```
@app.get("/hello")
@app.post("/echo")
```

Now let's add:

📘 PHASE 2: OpenAPI Spec & Swagger UI

This will enable:

- `/openapi.json`: live schema
- `/docs`: Swagger UI interface

🧩 Step 1: Store route metadata

We'll collect function info to build OpenAPI dynamically.

Update `MiniAPI`:

```
import json
from typing import Callable, Dict, List, Any
```

```
class MiniAPI:
    def __init__(self):
        self.routes: Dict[str, Dict[str, Callable]] = {}
        self.openapi_routes: List[Dict[str, Any]] = []

    def route(self, path: str, method: str, summary: str = ""):
        def decorator(func: Callable):
            self.routes.setdefault(path, {})[method] = func
            self.openapi_routes.append({
                "path": path,
                "method": method.lower(),
                "summary": summary or func.__name__,
            })
            return func
        return decorator

    def get(self, path: str, summary: str = ""):
        return self.route(path, "GET", summary)

    def post(self, path: str, summary: str = ""):
        return self.route(path, "POST", summary)

    async def __call__(self, scope, receive, send):
        assert scope["type"] == "http"
        path = scope["path"]
        method = scope["method"]

        if path == "/openapi.json":
            spec = {
                "openapi": "3.0.0",
                "info": {"title": "MiniAPI", "version": "1.0"},
                "paths": {},
            }
            for r in self.openapi_routes:
                p = r["path"]
                m = r["method"]
                spec["paths"].setdefault(p, {})[m] = {
                    "summary": r["summary"],
                    "responses": {
                        "200": {"description": "Success"}
                    }
                }
            await send({
```

```
        "type": "http.response.start",
        "status": 200,
        "headers": [(b"content-type", b"application/json")],
    })
    await send({
        "type": "http.response.body",
        "body": json.dumps(spec).encode(),
    })
    return

if path == "/docs":
    html = f"""
    <!DOCTYPE html>
    <html>
    <head>
        <title>Swagger UI</title>
        <link rel="stylesheet" href="https://unpkg.com/swagger-ui-dist/swagger-ui.css" />
    </head>
    <body>
        <div id="swagger-ui"></div>
        <script src="https://unpkg.com/swagger-ui-dist/swagger-ui-bundle.js"></script>
        <script>
        SwaggerUIBundle({{
            url: '/openapi.json',
            dom_id: '#swagger-ui'
        }});
        </script>
    </body>
    </html>
    """
    await send({
        "type": "http.response.start",
        "status": 200,
        "headers": [(b"content-type", b"text/html")],
    })
    await send({
        "type": "http.response.body",
        "body": html.encode(),
    })
    return

# Regular GET/POST handling like before
if path in self.routes and method in self.routes[path]:
    func = self.routes[path][method]
```

```
            if method == "POST":
                body = b""
                while True:
                    message = await receive()
                    body += message.get("body", b"")
                    if not message.get("more_body", False):
                        break
                data = json.loads(body.decode())
                response = await func(data)
            else:
                response = await func()

            await send({
                "type": "http.response.start",
                "status": 200,
                "headers": [(b"content-type", b"application/json")],
            })
            await send({
                "type": "http.response.body",
                "body": json.dumps(response).encode(),
            })
        else:
            await send({
                "type": "http.response.start",
                "status": 404,
                "headers": [(b"content-type", b"text/plain")],
            })
            await send({
                "type": "http.response.body",
                "body": b"Not Found",
            })
```

✅ Example App

```
app = MiniAPI()

@app.get("/", summary="Homepage")
async def homepage():
    return {"hello": "world"}

@app.post("/echo", summary="Echo JSON")
async def echo(body: dict):
    return {"you_sent": body}
```

▶️ Try It!

```
uvicorn main:app --reload
```

Open:

- http://localhost:8000/docs → Swagger UI 🧙
- http://localhost:8000/openapi.json → Raw schema

🔌 Up Next: WebSocket Support? (optional)

WebSocket adds real-time capability.
Do you want to add that next, or move to packaging (`miniapi`)?

🟢 Option A: Add WebSocket (`/ws`)
📦 Option B: Package this as `miniapi` framework (`pip install -e .` ready)
🛡️ Option C: Add validation (like Pydantic)

Let me know!

Awesome! Let's continue building our **MiniAPI microframework** with support for `@get()` and `@post()` decorators—just like FastAPI. This will allow us to register route handlers easily.

✅ Goal: Add `@get()` and `@post()` Decorators

We'll update our ASGI-based framework to support route registration like this:

```
@get("/hello")
def say_hello():
    return {"message": "Hi from GET"}

@post("/submit")
```

```
async def submit_data(data: dict):
    return {"received": data}
```

🔧 Step-by-Step Implementation

We'll do this in three parts:

📁 1. Core `MiniAPI` with Decorator Support

```
import json
from typing import Callable
from urllib.parse import parse_qs

# Base MiniAPI framework
class MiniAPI:
    def __init__(self):
        self.routes = {"GET": {}, "POST": {}}

    def __call__(self, scope, receive, send):
        assert scope["type"] == "http"
        method = scope["method"]
        path = scope["path"]

        handler = self.routes.get(method, {}).get(path)
        if not handler:
            return self._response(send, 404, {"error": "Not Found"})

        async def run():
            if method == "POST":
                # read and parse body
                body = await receive()
                raw_body = body.get("body", b"")
                try:
                    data = json.loads(raw_body.decode("utf-8"))
                except Exception:
                    data = None
                resp = await handler(data) if callable(handler) else {"error": "Invalid"}
            else:
                resp = handler()

            await self._response(send, 200, resp)
        return run()

    async def _response(self, send, status_code, data):
        await send({
            "type": "http.response.start",
            "status": status_code,
            "headers": [
                (b"content-type", b"application/json"),
            ]
        })
        await send({
            "type": "http.response.body",
            "body": json.dumps(data).encode("utf-8"),
        })

    def get(self, path: str):
        def decorator(func: Callable):
            self.routes["GET"][path] = func
            return func
        return decorator

    def post(self, path: str):
        def decorator(func: Callable):
            async def async_handler(data):
                return await func(data) if callable(func) else {}
            self.routes["POST"][path] = async_handler
            return func
        return decorator
```

🧪 2. Example Usage

```
app = MiniAPI()

@app.get("/hello")
def hello():
    return {"message": "Hello World"}

@app.post("/echo")
async def echo(data):
    return {"you_sent": data}
```

🚀 3. Run It Using Uvicorn

Save this as `miniapi.py` and run:

uvicorn miniapi:app --reload

Visit:

- `http://127.0.0.1:8000/hello` for GET
- `POST` to `http://127.0.0.1:8000/echo` with JSON using Postman/curl

✅ What's Working

- `@get(path)` decorator for registering GET routes.
- `@post(path)` for async POST handlers with JSON parsing.
- Barebones ASGI server behavior.

🧭 What’s Next?

Now that we have decorators, next steps are:

1. 📘 Generate OpenAPI Spec & Swagger UI
2. 🔌 Add WebSocket support (optional)
3. 🧰 Package all into a `miniapi` module like FastAPI

Shall we move to **Swagger + OpenAPI auto-generation**, so you can view docs like in FastAPI?

Or want to try WebSocket support first for fun?

Let me know.

Great! You're at an exciting point. Let's clarify your options and proceed step-by-step.

🚦 Your Options

🟢 Option A: Add WebSocket support

- Adds real-time communication (chat, notifications, etc.)
- We'll implement raw ASGI WebSocket handling — no FastAPI/Starlette.

📦 Option B: Package everything as `miniapi` framework

- Structure your toy framework like a real pip-installable package.
- You'll get:
  - `miniapi` module (with routing, request handling)
  - CLI or startup script (`run.py`)
  - Editable install (`pip install -e .`)

🛡️ Option C: Add validation (like Pydantic)

- Add type validation and parsing of query/body like FastAPI
- You'll create your own version of how FastAPI does this using:
  - Python type hints
  - Manual parsing or libraries like `pydantic-core` or `valida`

🧠 Recommendation

Since you've already explored:

- raw ASGI
- routing (`@get`, `@post`)
- response handling

👉 **Let’s add WebSocket support next**, because:

- It completes your understanding of ASGI protocol
- It’s harder to understand in FastAPI unless you see it raw
- It’s a fun and powerful thing to implement manually!

✅ Step-by-Step Plan: Add WebSocket Support

1. Extend your ASGI `app` to support WebSocket connections

We’ll check `scope['type'] == 'websocket'` and handle lifecycle events:

- `websocket.connect`
- `websocket.receive`
- `websocket.send`
- `websocket.disconnect`

💡 Are you ready to build `/ws` WebSocket endpoint manually in ASGI?

If yes, I’ll give you the raw working code and explain every line.

✅ Say “yes” to start!

Perfect! Let’s build **raw WebSocket support** into your ASGI app — no Starlette, no FastAPI, just pure ASGI.

🛠️ Step-by-Step: Add WebSocket Support to Your ASGI App

✅ Goal:

Create a `/ws` endpoint that:

- Accepts WebSocket connections
- Echoes back messages sent by the client
- Closes the connection gracefully

🧪 Raw ASGI Code with WebSocket Handling

```
# asgi_app.py
from typing import Callable

async def app(scope, receive, send):
    if scope['type'] == 'http':
        await send({
            "type": "http.response.start",
            "status": 200,
            "headers": [
                [b"content-type", b"text/plain"]
            ]
        })
        await send({
            "type": "http.response.body",
            "body": b"Hello from HTTP!",
        })

    elif scope['type'] == 'websocket':
        path = scope['path']

        if path == "/ws":
            await send({"type": "websocket.accept"})

            while True:
                message = await receive()

                if message["type"] == "websocket.receive":
                    text = message.get("text", "")
                    print(f"Received via WS: {text}")

                    # Echo back the message
                    await send({
                        "type": "websocket.send",
                        "text": f"Echo: {text}"
                    })

                elif message["type"] == "websocket.disconnect":
                    print("WebSocket disconnected")
                    break
        else:
            # Reject unknown WebSocket paths
            await send({"type": "websocket.close"})
```

🚀 How to Run

You need an ASGI server like `uvicorn`:

```
uvicorn asgi_app:app --reload
```

💬 How to Test WebSocket?

Use browser or Postman alternatives:

Option A: WebSocket King Client (Chrome)

- Connect to `ws://localhost:8000/ws`
- Send a message → get echo back!

Option B: JavaScript in browser dev console

```
const ws = new WebSocket("ws://localhost:8000/ws");
ws.onmessage = (msg) => console.log("Server:", msg.data);
ws.onopen = () => ws.send("Hello, server!");
```

🧠 What You Just Learned

- ASGI's `scope`, `receive`, `send` works for both HTTP and WebSocket
- WebSocket uses:
  - `websocket.accept`
  - `websocket.receive`
  - `websocket.send`
  - `websocket.disconnect`

🧭 Next Step Options

🔁 A: Add routing system for WebSocket (like `/ws/chat`, `/ws/notify`)

📦 B: Package into `miniapi` module (framework style)

🛡️ C: Add validation and type casting

What do you want to do next?

Great question — and no worries. Let’s clear it up simply:

⚙️ What is `uvicorn`?

`uvicorn` is an **ASGI server** — it's the program that **runs** your FastAPI (or any ASGI) app, just like:

- `gunicorn` runs WSGI apps (like Flask, Django pre-3.0)
- `node` runs JavaScript apps
- `python script.py` runs a script

🧠 Why Do You Need `uvicorn`?

Because **FastAPI** and **ASGI apps** are just **Python code** — they don't run by themselves.

`uvicorn`:

- Handles incoming **HTTP/WebSocket** connections
- Talks to your `app(scope, receive, send)` interface (ASGI)
- Runs it **asynchronously** (very fast)

📦 Install It

```
pip install uvicorn
```

🚀 How to Run Your App with `uvicorn`

Assume your app is in `main.py` and your app instance is called `app`:

```
uvicorn main:app --reload
```

Breakdown:

- `main` → filename (without `.py`)

- `app` → the ASGI app inside `main.py`
- `--reload` → auto-reload when code changes (great for development)

🧬 ASGI vs WSGI Recap

Feature	WSGI (Flask, Django old)	ASGI (FastAPI, Starlette)
Sync only	✅	❌
Async support	❌	✅
WebSockets	❌	✅
Server	gunicorn	uvicorn / daphne / hypercorn

🔥 Summary

> `uvicorn` is to FastAPI what `gunicorn` is to Flask.
> It's the fast, async web server that powers your app.

Let me know if you want to go deeper or even see how `uvicorn` works internally!

Awesome! Let's demystify how **`uvicorn` works internally** — and we'll do it step by step, beginner-friendly, but deep enough to satisfy your curiosity. 🚀

🔧 1. The Core Idea: ASGI Spec

First, Uvicorn is built to run **ASGI apps**.

> **ASGI** = Asynchronous Server Gateway Interface
> It's a **standard** that defines how a web server talks to an async Python app.

An ASGI app is a **callable like this**:

```
async def app(scope, receive, send):
    ...
```

Where:

- `scope` → Info about the connection (method, headers, path)
- `receive()` → Waits for the next event (like request body)
- `send()` → Sends response data

So **Uvicorn runs this function** with the right data.

🧱 2. What Does Uvicorn Actually Do?

✅ Uvicorn:

- **Opens a socket** and listens on a port (e.g., `127.0.0.1:8000`)
- **Accepts HTTP or WebSocket requests**
- **Parses** the HTTP protocol (via an HTTP parser)
- **Converts** it into an ASGI `scope`
- **Calls your ASGI app** with `scope`, `receive`, and `send`
- **Waits for your app to `send` a response**
- **Sends the response** back to the client

⚙️ 3. Internals (Architecture)

Uvicorn is structured like this:

```
uvicorn/
├── config.py        # Parses command line, loads app
├── main.py          # CLI entry point (uvicorn app:app)
├── server.py         # Main server loop
├── protocols/        # Handles HTTP, WebSocket, etc.
│   └── http/h11_impl.py  # Uses h11 to parse HTTP/1.1
├── loops/            # Async I/O backends: asyncio, uvloop
├── workers.py        # For gunicorn worker class
```

- **`uvicorn.protocols.http.h11_impl`** → Uses h11, a pure Python HTTP parser.
- **`uvicorn.loops`** → You can choose between:
    - `asyncio` (default Python async)
    - `uvloop` (ultra-fast, powered by libuv, same as Node.js)

🧪 4. What Happens When You Run:

```
uvicorn main:app --reload
```

Here's what happens:

1. **Loads your app**
   Parses `main:app` → imports `main.py`, gets `app`
2. **Creates the server**
   Starts an event loop, sets up sockets
3. **Accepts a connection**
   Parses the HTTP request via `h11`

Creates ASGI scope
A dictionary like:

```
{
 "type": "http",
 "method": "GET",
 "path": "/",
 "headers": [...],
 ...
}
```

4.

Calls your app
Passes it to your FastAPI app like:

```
await app(scope, receive, send)
```

5.
6. **Gets response**
   Waits for your app to `send` response chunks

7. **Sends it back**
   Writes it to the socket and closes the connection

🔍 Example: Your ASGI App

```
async def app(scope, receive, send):
    if scope["type"] == "http":
        await receive()  # Optional (if there's a body)
        await send({
            "type": "http.response.start",
            "status": 200,
            "headers": [(b"content-type", b"text/plain")]
        })
        await send({
            "type": "http.response.body",
            "body": b"Hello, world!",
        })
```

Uvicorn runs this `app()` when a request comes in.

🛠️ Bonus: Try Uvicorn from Python

You can also launch uvicorn programmatically:

```
import uvicorn
```

```
uvicorn.run("main:app", host="127.0.0.1", port=8000, reload=True)
```

🔥 Summary

Thing	What it does
`uvicorn`	ASGI server, runs your FastAPI app
`h11`	Low-level HTTP/1.1 parser
`uvloop`	Fast event loop backend (optional)
`scope`	Connection metadata
`receive`/`send`	Communication between server & app

Let me know if you want to build a **mini-uvicorn** for fun (tiny ASGI runner), or dive into **WebSocket support**, **gunicorn workers**, or **uvicorn performance tuning**.