AI VOID FIELD GUIDE

Step-by-Step Tutorials

Focused, practical tutorials that teach you how to accomplish specific tasks step by step. Learn by doing with clear instructions and working code examples.

AI VOID

2026-05-26 20:32 IST

Contents

CHAPTER 01

Build a REST API with FastAPI

What you'll build: A functional REST API using FastAPI, demonstrating setup, route definition, data validation, dependency injection, and testing. **Time needed:** ~75 minutes **Prerequisites:** Python 3.8 or newer, Basic understanding of Python, Familiarity with REST API concepts **Version used:** 0.115.6

FastAPI is a modern, fast (high-performance) web framework for building APIs with Python 3.8+ based on standard Python type hints. It's designed to be easy to use, highly performant, and automatically generate interactive API documentation. In this tutorial, we'll walk through building a complete, yet simple, REST API from scratch, covering the core features that make FastAPI a joy to work with.

Prerequisites and Environment Setup

Before we dive into coding, we need to ensure your development environment is ready. This involves having Python installed and setting up a dedicated virtual environment for our project. Using a virtual environment helps keep your project's dependencies isolated from other Python projects you might have.

Step 1: Verify Python Installation

First, let's confirm you have Python 3.8 or newer installed on your system. Open your terminal or command prompt and run:

```
python3 --version
```

You should see an output like `Python 3.X.X` where `X.X` is `8` or higher. If you don't have Python installed or have an older version, please install the latest Python 3 from the official Python website.

Step 2: Create a Project Directory

Next, create a new directory for our FastAPI project. This will keep all our project files organized.

```
mkdir fastapi-api-tutorial
cd fastapi-api-tutorial
```

Step 3: Set Up a Virtual Environment

Inside your new project directory, let's create and activate a virtual environment.

```
python3 -m venv .venv
```

This command creates a new directory named `.venv` (you can name it anything, but `.venv` is common) containing a fresh Python installation and a `pip` installer.

Now, activate the virtual environment:

On macOS/Linux:

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

On Windows (Command Prompt):

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

On Windows (PowerShell):

```
.venv\Scripts\Activate.ps1
```

You'll know it's active because your terminal prompt will usually show `(.venv)` or similar before your current path.

Step 4: Install FastAPI and Uvicorn

With your virtual environment active, we can now install FastAPI and Uvicorn. Uvicorn is an ASGI server that FastAPI uses to run your application. We'll install `fastapi` with the `[all]` extra to include common dependencies like `uvicorn` and `pydantic` automatically.

```
pip install "fastapi[all]"
```

This command installs FastAPI, Uvicorn (our server), and Pydantic (for data validation and serialization, which FastAPI uses extensively).

> ⚡ Note: The quotes around `fastapi[all]` are important, especially on some shells (like `zsh`) to prevent the `[` and `]` characters from being interpreted as special shell characters.

Step 5: Verify Installations

You can quickly check if `fastapi` and `uvicorn` are installed by listing the installed packages:

```
pip freeze
```

You should see `fastapi`, `uvicorn`, `pydantic`, and their dependencies in the list.

At this point, your environment is fully set up, and you're ready to start building your FastAPI application. You have a dedicated space for your project and all the necessary tools installed.

Initialize FastAPI Application

Now that our environment is ready, let's create the foundational file for our FastAPI application. This file will contain the main application instance and will be where we define all our API endpoints.

Step 1: Create `main.py`

In your `fastapi-api-tutorial` directory, create a new file named `main.py`. This is a common convention for the main entry point of a Python application.

```python
# main.py
from fastapi import FastAPI

# Create a FastAPI application instance
app = FastAPI()

# This is our first, very basic, route.
# We'll add more meaningful ones soon!
@app.get("/")
async def read_root():
    return {"message": "Hello World"}
```

Step 2: Understand the Code

Let's break down these few lines:

- `from fastapi import FastAPI`: We import the `FastAPI` class from the `fastapi` library. This class provides all the functionality to create your API.
- `app = FastAPI()`: This line creates an instance of the `FastAPI` class. This `app` object is the core of your API. It will be used to define all your API endpoints, handle requests, and manage application settings.
- `@app.get("/")`: This is a decorator. In Python, decorators are functions that modify other functions. Here, `@app.get("/")` tells FastAPI that the function immediately below it (`read_root`) should handle HTTP GET requests to the root URL path (`/`).
- `async def read_root():`: This defines an asynchronous function. FastAPI is built on asynchronous Python, which allows it to handle many requests concurrently without blocking. We'll explore `async` more later, but for now, know that most of your endpoint functions will be `async def`.
- `return {"message": "Hello World"}`: This function simply returns a Python dictionary. FastAPI automatically converts this dictionary into a JSON response, which is the standard format for REST APIs.

Step 3: Run Your FastAPI Application

Now, let's run our application using Uvicorn. Uvicorn is an ASGI server that will serve our FastAPI app.

Open your terminal (with the virtual environment activated) in the `fastapi-api-tutorial` directory and run the following command:

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

Let's dissect this command:

- `uvicorn`: This invokes the Uvicorn server.
- `main:app`: This tells Uvicorn where to find your FastAPI application. `main` refers to the `main.py` file, and `app` refers to the `FastAPI()` instance we created inside that file.
- `--reload`: This is an incredibly useful flag for development. It tells Uvicorn to automatically restart the server whenever you make changes to your code. This means you don't have to manually stop and start the server every time you save a file.

You should see output similar to this:

```
INFO:     Will watch for changes in these directories: ['/path/to/fastapi-api-
tutorial']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [xxxxx] using stat reload
INFO:     Started server process [xxxxx]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

The server is now running! Open your web browser and navigate to `<http://127.0.0.1:8000>`. You should see the JSON response: `{"message": "Hello World"}`.

> ⚠️ Common mistake: If you forget to activate your virtual environment, `uvicorn` might not be found, or it might run an older version installed globally. Always ensure `(.venv)` is in your prompt before running `uvicorn`.

You've successfully initialized your FastAPI application and run it locally. This is the foundation upon which we'll build all our API functionality.

Create Your First API Endpoint (GET)

We've already created a basic GET endpoint for the root path (`/`). Now, let's expand on that by creating another GET endpoint that simulates fetching a list of items. This will demonstrate how to define more specific routes and return data.

Step 1: Define Some Dummy Data

In a real application, this data would come from a database. For our tutorial, we'll use a simple Python list of dictionaries to represent our "items." Add this list to your `main.py` file, preferably above your `app = FastAPI()` line or just below it.

```python
# main.py
from fastapi import FastAPI

app = FastAPI()

# Dummy data for our items
items_db = [
    {"item_name": "Foo"},
    {"item_name": "Bar"},
    {"item_name": "Baz"},
]

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

```
async def read_root():
    return {"message": "Hello World"}
```

Step 2: Create a GET Endpoint for All Items

Now, let's add a new GET endpoint that will return our entire `items_db` list. We'll define this at the `/items/` path.

```
# main.py
from fastapi import FastAPI

app = FastAPI()

items_db = [
    {"item_name": "Foo"},
    {"item_name": "Bar"},
    {"item_name": "Baz"},
]

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

# New endpoint to get all items
@app.get("/items/")
async def read_items():
    return items_db
```

Step 3: Understand the New Endpoint

- `@app.get("/items/")`: This decorator registers the `read_items` function to handle GET requests specifically to the `/items/` URL path.
- `async def read_items():`: This is our asynchronous function that will be executed when a request hits this endpoint.
- `return items_db`: Just like before, FastAPI takes this Python list of dictionaries and automatically serializes it into a JSON array in the HTTP response.

Step 4: Verify the New Endpoint

Since you're running Uvicorn with the `--reload` flag, your server should have automatically restarted after you saved `main.py`.

Open your web browser or use a tool like `curl` to visit `<http://127.0.0.1:8000/items/>`.

You should see the JSON response:

```
[
  {
```

```
    "item_name": "Foo"
  },
  {
    "item_name": "Bar"
  },
  {
    "item_name": "Baz"
  }
]
```

You've now successfully created a second GET endpoint that serves a list of data. This demonstrates how to define distinct routes for different resources in your API.

Handle Request Bodies with Pydantic (POST)

Most APIs need to receive data from clients, especially for creating new resources. This data is typically sent in the request body of a POST or PUT request. FastAPI uses Pydantic models to define the structure and validation rules for these request bodies, making data handling robust and straightforward.

Step 1: Define a Pydantic Model

First, let's define the structure of an "Item" that clients can create. We'll use Pydantic's `BaseModel` for this. Add the following code to your `main.py` file, typically at the top after your imports.

```python
# main.py
from typing import Optional
from fastapi import FastAPI
from pydantic import BaseModel # Import BaseModel

# Define a Pydantic model for an Item
class Item(BaseModel):
    name: str
    description: Optional[str] = None # Optional string, defaults to None
    price: float
    tax: Optional[float] = None # Optional float, defaults to None

app = FastAPI()

items_db = [
    {"item_name": "Foo"},
    {"item_name": "Bar"},
    {"item_name": "Baz"},
]

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

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

```
async def read_items():
    return items_db
```

Let's break down the `Item` model:

- `from pydantic import BaseModel`: We import `BaseModel`, the base class for creating data models.
- `class Item(BaseModel):`: Our `Item` class inherits from `BaseModel`, making it a Pydantic model.
- `name: str`: This defines a required field `name` that must be a string.
- `description: Optional[str] = None`: This defines an optional field `description` that must be a string. If not provided, it defaults to `None`. `Optional` comes from Python's `typing` module.
- `price: float`: A required field `price` that must be a floating-point number.
- `tax: Optional[float] = None`: Another optional float field `tax`.

FastAPI uses these type hints to perform automatic data validation, serialization, and to generate your API documentation.

Step 2: Create a POST Endpoint to Create an Item

Now, let's create an endpoint that accepts an `Item` object in the request body.

```
# main.py
from typing import Optional
from fastapi import FastAPI
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None

app = FastAPI()

# A simple in-memory "database" to store items
# We'll use a dictionary for easy lookup by ID later
in_memory_items = {}
next_item_id = 1

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

@app.get("/items/")
async def read_items():

# For now, just return our dummy items. We'll integrate the Pydantic items
```

```
later.
    return items_db

# New POST endpoint to create an item
@app.post("/items/")
async def create_item(item: Item):
    global next_item_id # Declare intent to modify the global variable
    item_id = next_item_id
    in_memory_items[item_id] = item.dict() # Store the item (converted to
dict)
    next_item_id += 1
    return {"item_id": item_id, **item.dict()} # Return the created item with
its ID
```

In this new `create_item` endpoint:

- `@app.post("/items/")`: This decorator registers the function to handle HTTP POST requests to the `/items/` path.
- `async def create_item(item: Item):`: This is where the magic happens. By declaring the `item` parameter with the type hint `Item` (our Pydantic model), FastAPI automatically:
    - Reads the request body as JSON.
    - Validates the data against the `Item` model's schema.
    - Converts the data into an `Item` object.
    - Provides helpful error messages if the data doesn't match the schema (e.g., `price` is not a number).
- `global next_item_id`: We need this to modify the `next_item_id` variable defined globally.
- `in_memory_items[item_id] = item.dict()`: We store the received `Item` in our simple in-memory dictionary. `item.dict()` converts the Pydantic model back into a standard Python dictionary.
- `return {"item_id": item_id, **item.dict()}`: We return the newly created item, including the assigned `item_id`, back to the client. The `**item.dict()` unpacks the item's attributes into the dictionary.

Step 3: Verify the POST Endpoint Using Interactive Docs

FastAPI automatically generates interactive API documentation based on your code and Pydantic models. This is incredibly useful for testing and understanding your API.

Ensure your Uvicorn server is running with `--reload`. If not, run `uvicorn main:app --reload` again.

Open your web browser and go to `<http://127.0.0.1:8000/docs>`.

You'll see the Swagger UI documentation.

1. Find the `POST /items/` endpoint and click on it to expand.
2. Click the "Try it out" button.
3. In the "Request body" field, you'll see an example JSON payload based on your `Item` model. Modify it to create a new item, for example:

```
    {
      "name": "My New Book",
      "description": "A thrilling adventure novel.",
      "price": 29.99,
      "tax": 2.50
    }
```

1. Click the "Execute" button.

You should see a `200 OK` response with the created item and its ID in the "Response body."

Try sending invalid data (e.g., `price` as a string) and observe the validation error messages FastAPI provides.

You've now successfully implemented a POST endpoint that handles request bodies using Pydantic for robust data validation and automatic documentation.

Implement Path and Query Parameters

APIs often need to retrieve or filter specific resources. **Path parameters** allow you to identify a specific resource directly in the URL (e.g., `/items/123` to get item with ID 123). **Query parameters** are used for optional filtering, sorting, or pagination, appended after a `?` in the URL (e.g., `/items/?skip=0&limit=10`).

Step 1: Add a Path Parameter for Retrieving a Single Item

Let's create an endpoint to fetch a single item by its ID. We'll use a path parameter for the `item_id`.

```
# main.py (updated sections)
from typing import Optional
from fastapi import FastAPI, HTTPException # Import HTTPException
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: Optional[str] = None
```

```
    price: float
    tax: Optional[float] = None

app = FastAPI()

in_memory_items = {}
next_item_id = 1

# ... (existing @app.get("/") and @app.get("/items/") endpoints)

@app.post("/items/")
async def create_item(item: Item):
    global next_item_id
    item_id = next_item_id
    in_memory_items[item_id] = item.dict()
    next_item_id += 1
    return {"item_id": item_id, **item.dict()}

# New GET endpoint with a path parameter
@app.get("/items/{item_id}")
async def read_item(item_id: int): # Declare item_id as an integer path
parameter
    if item_id not in in_memory_items:
        raise HTTPException(status_code=404, detail="Item not found")
    return in_memory_items[item_id]
```

Here's what's new:

- `@app.get("/items/{item_id}")`: The `{item_id}` in the path indicates a path parameter. FastAPI will capture the value at this position in the URL.
- `async def read_item(item_id: int):`: By declaring `item_id` with a type hint of `int`, FastAPI automatically:
    - Validates that the value provided in the URL path is an integer.
    - Converts it to a Python `int`.
    - Provides clear error messages if the type is incorrect (e.g., `/items/abc`).
- `if item_id not in in_memory_items: raise HTTPException(status_code=404, detail="Item not found")`: This is basic error handling. If the item ID doesn't exist in our `in_memory_items` dictionary, we raise an `HTTPException` with a 404 Not Found status code. We'll cover error handling in more detail later.

Step 2: Add Query Parameters for Filtering/Pagination

Now, let's modify our existing `/items/` GET endpoint to support query parameters for pagination (skipping a number of items and limiting the results).

```
# main.py (updated @app.get("/items/") endpoint)
# ... (imports and Item model)
```

```
app = FastAPI()

in_memory_items = {}
next_item_id = 1

# ... (existing @app.get("/") endpoint)

# Updated GET endpoint with query parameters
@app.get("/items/")
async def read_items_with_pagination(skip: int = 0, limit: int = 10): # Query
parameters with default values
    # Convert dictionary values to a list to apply skip/limit
    items_list = list(in_memory_items.values())
    return items_list[skip : skip + limit]

# ... (existing @app.post("/items/") and @app.get("/items/{item_id}")
endpoints)
```

In the updated `read_items_with_pagination` endpoint:

- `skip: int = 0, limit: int = 10`: These are our query parameters.
  - By giving them default values (`= 0`, `= 10`), they become optional. If the client doesn't provide them, these defaults are used.
  - FastAPI automatically validates their types (`int`) and provides conversion.
- `items_list[skip : skip + limit]`: This Python slice operation applies the pagination logic to our list of items.

> ⚡ Note: We renamed `read_items` to `read_items_with_pagination` for clarity, but the URL path remains `/items/`.

Step 3: Verify Path and Query Parameters

Ensure your Uvicorn server is running.

1. **Test Path Parameter:**
    - First, use the `POST /items/` endpoint in `/docs` to create a few items. Note down their `item_id`s (e.g., 1, 2, 3).
    - Now, in your browser, go to `<http://127.0.0.1:8000/items/1>` (replace `1` with an actual ID). You should see the JSON for that specific item.
    - Try `<http://127.0.0.1:8000/items/999>` (an ID that doesn't exist). You should get a `404 Not Found` error.
    - Try `<http://127.0.0.1:8000/items/notanumber>`. You'll get a validation error from FastAPI, indicating the `item_id` must be an integer.
2. **Test Query Parameters:**
    - Go to `<http://127.0.0.1:8000/items/>`. You'll see the first 10 items (or fewer if you haven't created that many).
    - Try `<http://127.0.0.1:8000/items/?skip=1&limit=1>`. This should return only the second item.
    - Experiment with different `skip` and `limit` values.

You've now successfully implemented both path and query parameters, allowing clients to request specific resources and filter lists of resources.

Add PUT and DELETE Endpoints

A complete REST API typically supports all CRUD (Create, Read, Update, Delete) operations. We've covered Create (POST) and Read (GET). Now, let's add Update (PUT) and Delete (DELETE) functionality.

Step 1: Implement a PUT Endpoint to Update an Item

The PUT method is used to update an existing resource. It typically takes a path parameter to identify the resource and a request body (similar to POST) with the updated data.

```
# main.py (updated sections)
from typing import Optional
from fastapi import FastAPI, HTTPException
```

```python
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None

app = FastAPI()

in_memory_items = {}
next_item_id = 1

# ... (existing GET and POST endpoints)

# New PUT endpoint to update an item
@app.put("/items/{item_id}")
async def update_item(item_id: int, item: Item):
    if item_id not in in_memory_items:
        raise HTTPException(status_code=404, detail="Item not found")

    # Update the item in our in-memory storage
    in_memory_items[item_id] = item.dict()
    return {"item_id": item_id, **item.dict()}
```

In this `update_item` endpoint:

- `@app.put("/items/{item_id}")`: This decorator handles HTTP PUT requests to a specific item ID.
- `async def update_item(item_id: int, item: Item):`: This function takes both a path parameter (`item_id: int`) and a request body (`item: Item`), leveraging FastAPI's automatic type validation and Pydantic model parsing for both.
- We again check if the `item_id` exists and raise a `404 HTTPException` if not.
- `in_memory_items[item_id] = item.dict()`: We simply overwrite the existing item's data with the new data from the request body. In a real application, you'd likely update specific fields rather than replacing the whole object, but for simplicity, we're replacing it here.

Step 2: Implement a DELETE Endpoint to Remove an Item

The DELETE method is used to remove a specific resource. It typically only requires a path parameter to identify the resource to be deleted.

```python
# main.py (updated sections)
from typing import Optional
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

class Item(BaseModel):
```

```
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None

app = FastAPI()

in_memory_items = {}
next_item_id = 1

# ... (existing GET, POST, and PUT endpoints)

# New DELETE endpoint to remove an item
@app.delete("/items/{item_id}")
async def delete_item(item_id: int):
    if item_id not in in_memory_items:
        raise HTTPException(status_code=404, detail="Item not found")

    del in_memory_items[item_id] # Remove the item from our storage
    return {"message": f"Item {item_id} deleted successfully"}
```

In this `delete_item` endpoint:

- `@app.delete("/items/{item_id}")`: This decorator handles HTTP DELETE requests.
- `async def delete_item(item_id: int):`: It takes the `item_id` as a path parameter.
- After checking for existence, `del in_memory_items[item_id]` removes the item from our dictionary.
- We return a confirmation message.

Step 3: Verify PUT and DELETE Endpoints

Ensure your Uvicorn server is running and open `<http://127.0.0.1:8000/docs>`.

1. **Test PUT:**
    - Use `POST /items/` to create an item (e.g., with ID 1) and note its ID.
    - Expand `PUT /items/{item_id}`.
    - Click "Try it out".
    - Enter the `item_id` (e.g., `1`).
    - Modify the request body to update the item, for example:

```
{
  "name": "Updated Book Title",
  "description": "A revised and improved adventure.",
  "price": 35.00,
  "tax": 3.00
```

```
        }
```

```
- Click "Execute". You should get a `200 OK` response with the updated item.
- Verify the update by using `GET /items/1`.
```

1. **Test DELETE:**
    - Use `POST /items/` to create another item (e.g., with ID 2).
    - Expand `DELETE /items/{item_id}`.
    - Click "Try it out".
    - Enter the `item_id` of the item you want to delete (e.g., `2`).
    - Click "Execute". You should get a `200 OK` response with the deletion message.
    - Verify the deletion by trying `GET /items/2`. You should now receive a `404 Not Found` error.

You've now added full CRUD capabilities to your API, handling updates and deletions with proper HTTP methods and path parameters.

Utilize Dependency Injection

Dependency Injection (DI) is a powerful pattern that FastAPI leverages extensively. It allows you to declare "dependencies" (functions or classes) that your endpoint functions need. FastAPI then automatically resolves and injects these dependencies when a request comes in. This promotes code reusability, testability, and better organization.

Step 1: Create a Simple Dependency

Let's imagine we have some common parameters that multiple endpoints might use, or a utility function that needs to be run before an endpoint. We'll create a simple dependency that provides common pagination parameters.

Add this function to your `main.py` file:

```python
# main.py (new function)
# ... (imports, Item model, app instance, in_memory_items)

# Define a dependency function
async def common_parameters(q: Optional[str] = None, skip: int = 0, limit:
int = 10):
    return {"q": q, "skip": skip, "limit": limit}
```

```
# ... (existing GET, POST, PUT, DELETE endpoints)
```

This `common_parameters` function simply defines three optional parameters: `q` (for a query string), `skip`, and `limit`. It returns them as a dictionary.

Step 2: Inject the Dependency into an Endpoint

Now, let's refactor our `read_items_with_pagination` endpoint to use this dependency. We'll import `Depends` from `fastapi`.

```
# main.py (updated imports and @app.get("/items/") endpoint)
from typing import Optional
from fastapi import FastAPI, HTTPException, Depends # Import Depends
from pydantic import BaseModel

# ... (Item model, app instance, in_memory_items, common_parameters function)

# Updated GET endpoint using dependency injection
@app.get("/items/")
async def read_items_with_dependency(commons: dict =
Depends(common_parameters)):
    # Convert dictionary values to a list to apply skip/limit
    items_list = list(in_memory_items.values())

    # Apply filtering based on 'q' if provided
    if commons["q"]:
        items_list = [item for item in items_list if commons["q"].lower() in i
tem.get("name", "").lower()]

    # Apply pagination
    return items_list[commons["skip"] : commons["skip"] + commons["limit"]]

# ... (existing @app.get("/{item_id}"), @app.post("/items/"), @app.put("/
items/{item_id}"), @app.delete("/items/{item_id}") endpoints)
```

What changed:

- `from fastapi import FastAPI, HTTPException, Depends`: We imported `Depends`.
- `commons: dict = Depends(common_parameters)`: This is the core of dependency injection.
    - We declare a parameter `commons` with a type hint `dict`.
    - We set its default value to `Depends(common_parameters)`. This tells FastAPI to call the `common_parameters` function, resolve its parameters from the request (query parameters in this case), and inject the returned value (the dictionary) into our `commons` variable.

- Inside the function, we now access `commons["skip"]`, `commons["limit"]`, and `commons["q"]` instead of directly accessing `skip` and `limit`.
- We've also added a simple filtering logic based on the `q` query parameter.

> ⚡ Note: This example shows a simple dependency that returns data. Dependencies can also perform actions like database connections, authentication checks, or permission validations.

Step 3: Verify Dependency Injection

Ensure your Uvicorn server is running.

Open `<http://127.0.0.1:8000/docs>`.

1. **Test the updated `/items/` endpoint:**
    - You'll notice that the `GET /items/` endpoint now shows `q`, `skip`, and `limit` as query parameters, just as defined in `common_parameters`.
    - Use `POST /items/` to create a few items with different names (e.g., "Apple", "Banana", "Orange").
    - Try `<http://127.0.0.1:8000/items/?skip=1&limit=1>`. This should still work as before.
    - Now, try `<http://127.0.0.1:8000/items/?q=apple>`. You should only see the "Apple" item.
    - Try `<http://127.0.0.1:8000/items/?q=an&skip=0&limit=1>`. This should show the first item containing "an" (e.g., "Banana").

Dependency injection significantly improves the modularity and maintainability of your FastAPI application by centralizing common logic and making your endpoint functions cleaner.

Implement Basic Error Handling

Even with robust data validation, things can go wrong: a resource might not be found, a user might not be authorized, or an internal server error could occur. Providing clear and consistent error responses is crucial for a good API experience. FastAPI makes this easy with `HTTPException`.

We've already used `HTTPException` a couple of times for `404 Not Found` errors. Let's review it and ensure our error handling is consistent.

Step 1: Consolidate Error Handling Logic

FastAPI's `HTTPException` is the primary way to raise HTTP-specific errors. When you raise an `HTTPException`, FastAPI catches it and returns a JSON response with the specified `status_code` and `detail` message.

Let's ensure all our endpoints that look up items handle the "not found" case gracefully.

```
# main.py (reviewing existing error handling)
from typing import Optional
from fastapi import FastAPI, HTTPException, Depends
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None

app = FastAPI()

in_memory_items = {}
next_item_id = 1

async def common_parameters(q: Optional[str] = None, skip: int = 0, limit:
int = 10):
    return {"q": q, "skip": skip, "limit": limit}

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

@app.get("/items/")
async def read_items_with_dependency(commons: dict =
Depends(common_parameters)):
    items_list = list(in_memory_items.values())
    if commons["q"]:
        items_list = [item for item in items_list if commons["q"].lower() in i
tem.get("name", "").lower()]
    return items_list[commons["skip"] : commons["skip"] + commons["limit"]]

@app.post("/items/")
async def create_item(item: Item):
    global next_item_id
    item_id = next_item_id
    in_memory_items[item_id] = item.dict()
    next_item_id += 1
    return {"item_id": item_id, **item.dict()}

# Endpoint with explicit 404 error handling
@app.get("/items/{item_id}")
async def read_item(item_id: int):
    if item_id not in in_memory_items:
        raise HTTPException(status_code=404, detail="Item not found") #
Explicit 404
    return in_memory_items[item_id]
```

```python
# Endpoint with explicit 404 error handling
@app.put("/items/{item_id}")
async def update_item(item_id: int, item: Item):
    if item_id not in in_memory_items:
        raise HTTPException(status_code=404, detail="Item not found") #
Explicit 404

    in_memory_items[item_id] = item.dict()
    return {"item_id": item_id, **item.dict()}

# Endpoint with explicit 404 error handling
@app.delete("/items/{item_id}")
async def delete_item(item_id: int):
    if item_id not in in_memory_items:
        raise HTTPException(status_code=404, detail="Item not found") #
Explicit 404

    del in_memory_items[item_id]
    return {"message": f"Item {item_id} deleted successfully"}
```

As you can see, we've already consistently applied `raise HTTPException(status_code=404, detail="Item not found")` in our `read_item`, `update_item`, and `delete_item` endpoints. This is the correct and idiomatic way to handle such errors in FastAPI.

Step 2: Custom Error Responses (Optional, but good to know)

For more advanced error handling, you can also define custom exception handlers. For instance, if you wanted to handle a specific custom exception or override FastAPI's default `HTTPException` response format.

Let's add a very simple custom exception and handler as an example. This isn't strictly necessary for "basic" error handling but shows the capability.

```python
# main.py (add custom exception and handler)
from typing import Optional
from fastapi import FastAPI, HTTPException, Depends, Request # Import Request
from fastapi.responses import JSONResponse # Import JSONResponse
from pydantic import BaseModel

# Define a custom exception
class CustomException(Exception):
    def __init__(self, name: str):
        self.name = name

# ... (Item model, app instance, in_memory_items, common_parameters)

app = FastAPI()

# Register a custom exception handler
@app.exception_handler(CustomException)
async def custom_exception_handler(request: Request, exc: CustomException):
    return JSONResponse(
        status_code=418, # I'm a teapot!
```

```
        content={"message": f"Oops! Custom error for: {exc.name}"},
    )

# ... (existing GET, POST, PUT, DELETE endpoints)

# Add a new endpoint to demonstrate the custom exception
@app.get("/teapot/")
async def demonstrate_custom_error():
    raise CustomException(name="Tea Time")
```

Here's what we added:

- `class CustomException(Exception):`: A simple custom Python exception.
- `@app.exception_handler(CustomException)`: This decorator registers an asynchronous function (`custom_exception_handler`) to specifically catch `CustomException` instances that are raised anywhere in your API.
- `JSONResponse(...)`: Inside the handler, we explicitly create a `JSONResponse` with a custom status code (418 "I'm a teapot" is a fun example) and a custom message.
- `@app.get("/teapot/")`: A new endpoint that simply raises our `CustomException`.

Step 3: Verify Error Handling

Ensure Uvicorn is running.

1. **Test `HTTPException` (404):**
    - Go to `<http://127.0.0.1:8000/items/999>` (assuming item 999 doesn't exist). You should get a JSON response like: `{"detail":"Item not found"}` with a 404 status code.
    - In `/docs`, try to `PUT` or `DELETE` an item with a non-existent ID. You'll see the same `404` response.
2. **Test Custom Exception (if implemented):**
    - Go to `<http://127.0.0.1:8000/teapot/>`. You should see the custom JSON response: `{"message":"Oops! Custom error for: Tea Time"}` with a 418 status code.

Consistent error handling makes your API predictable and easier for clients to integrate with. By using `HTTPException` and optionally custom handlers, you can provide clear feedback when things don't go as expected.

Test Your API Endpoints

Writing tests is a critical part of building robust applications. FastAPI provides an excellent `TestClient` utility that allows you to simulate requests to your API without actually running the Uvicorn server, making your tests fast and reliable. We'll use `pytest`, a popular Python testing framework.

Step 1: Install `pytest`

First, if you haven't already, install `pytest` in your virtual environment.

```
pip install pytest
```

Step 2: Create a Test File

Create a new file named `test_main.py` in your `fastapi-api-tutorial` directory.

```
# test_main.py
from fastapi.testclient import TestClient
from main import app, in_memory_items, next_item_id # Import app and our
"database"

# Create a TestClient instance for our FastAPI app
client = TestClient(app)

# Helper function to reset our in-memory "database" before each test
def setup_function():
    global in_memory_items, next_item_id
    in_memory_items = {}
    next_item_id = 1

# Test for the root endpoint
def test_read_root():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Hello World"}

# Test for creating an item (POST)
def test_create_item():
    setup_function() # Reset database for a clean test
    response = client.post(
        "/items/",
        json={"name": "Test Item", "description": "A test description", "price
": 10.0, "tax": 1.0}
    )
    assert response.status_code == 200
    assert response.json()["name"] == "Test Item"
    assert response.json()["item_id"] == 1

# Test for reading a specific item (GET with path parameter)
def test_read_item():
```

```
    setup_function()
    # First, create an item so we have something to read
    client.post("/items/", json={"name": "Read Item", "price": 20.0})

    response = client.get("/items/1")
    assert response.status_code == 200
    assert response.json()["name"] == "Read Item"

# Test for reading a non-existent item (GET with path parameter, expecting
404)
def test_read_non_existent_item():
    setup_function()
    response = client.get("/items/999")
    assert response.status_code == 404
    assert response.json() == {"detail": "Item not found"}

# Test for updating an item (PUT)
def test_update_item():
    setup_function()
    client.post("/items/", json={"name": "Original Item", "price": 5.0})
    response = client.put(
        "/items/1",
        json={"name": "Updated Item", "description": "Updated desc", "price":
15.0, "tax": 1.5}
    )
    assert response.status_code == 200
    assert response.json()["name"] == "Updated Item"
    assert response.json()["price"] == 15.0

# Test for deleting an item (DELETE)
def test_delete_item():
    setup_function()
    client.post("/items/", json={"name": "Item to Delete", "price": 25.0})
    response = client.delete("/items/1")
    assert response.status_code == 200
    assert response.json() == {"message": "Item 1 deleted successfully"}

    # Verify it's actually gone
    get_response = client.get("/items/1")
    assert get_response.status_code == 404
```

Step 3: Understand the Test Code

- `from fastapi.testclient import TestClient`: This imports the `TestClient`, which is a wrapper around `httpx` (an HTTP client library) that allows you to make requests to your FastAPI application directly.
- `from main import app, in_memory_items, next_item_id`: We import our `FastAPI` app instance and our "database" variables (`in_memory_items`, `next_item_id`) so we can manipulate and reset them for isolated tests.
- `client = TestClient(app)`: This creates an instance of the `TestClient`, passing our FastAPI `app` object to it. Now `client` can make requests.

- `def setup_function():`: This function is a `pytest` hook that runs before each test function. We use it to reset our `in_memory_items` and `next_item_id` to ensure each test starts with a clean slate, preventing tests from interfering with each other.
- `def test_read_root():`: Pytest automatically discovers functions starting with `test_`.
- `response = client.get("/")`: We make a GET request to the root path. The `TestClient` methods (`.get()`, `.post()`, `.put()`, `.delete()`) mimic actual HTTP requests.
- `assert response.status_code == 200`: We assert that the HTTP status code in the response is 200 (OK).
- `assert response.json() == {"message": "Hello World"}`: We assert that the JSON content of the response matches our expected dictionary.

Step 4: Run Your Tests

Make sure your Uvicorn server is not running when you run tests, as the `TestClient` interacts with the application directly in memory.

In your terminal (with the virtual environment activated), run `pytest`:

```
pytest
```

You should see output indicating that all your tests passed:

```
============================= test session starts
==============================
platform linux -- Python 3.x.x, pytest-x.x.x, pluggy-x.x.x
rootdir: /path/to/fastapi-api-tutorial
collected 6
items

test_main.py ......
[100%]

============================== 6 passed in x.xxs
===============================
```

> ⚠️ Common mistake: Forgetting `setup_function()` or similar cleanup for in-memory databases can lead to "flaky" tests where tests pass or fail depending on the order they run. Always reset your state for each test.

You've now successfully implemented unit tests for your FastAPI endpoints using `TestClient` and `pytest`, ensuring your API behaves as expected.

Run the API and Explore Interactive Docs

You've built a functional REST API, and now it's time to appreciate the developer experience FastAPI offers, particularly its automatic interactive documentation.

Step 1: Start Your FastAPI Application

If you stopped your Uvicorn server for testing, start it again now. Ensure your virtual environment is active.

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

You should see the "Uvicorn running on `<http://127.0.0.1:8000`"> message.

Step 2: Explore the Swagger UI Docs

Open your web browser and navigate to `<http://127.0.0.1:8000/docs`>.

This page presents the Swagger UI, a dynamically generated interactive documentation for your API.

- **Automatic Generation:** Notice how all the endpoints you defined (GET, POST, PUT, DELETE) are listed, along with their HTTP methods, paths, and descriptions.
- **Pydantic Integration:** The request bodies for POST and PUT endpoints, as well as the response models, are automatically derived from your Pydantic `Item` model, showing required fields, types, and default values.
- **Try It Out:** For each endpoint, you can click "Try it out," fill in parameters (path, query, body), and click "Execute." The UI will send an actual request to your running API and display the response, including status code, response body, and headers. This is incredibly useful for quickly testing your API without external tools like Postman or curl.
- **Schema Definitions:** Scroll down to see the "Schemas" section, which clearly defines your `Item` model.

Step 3: Explore the ReDoc Docs

FastAPI also automatically generates alternative documentation using ReDoc. Open a new tab in your browser and go to `<http://127.0.0.1:8000/redoc`>.

ReDoc provides a different, more compact, and often more readable layout for API documentation, especially for larger APIs. It's another view of the same underlying OpenAPI specification that FastAPI generates.

Step 4: Understand the Value of Automatic Documentation

The automatic generation of OpenAPI (formerly Swagger) documentation is one of FastAPI's most powerful features:

- **Developer Productivity:** You don't have to manually write and maintain API documentation, which often becomes outdated. FastAPI keeps it in sync with your code.
- **Client Integration:** Other developers consuming your API can easily understand how to use it, what data to send, and what responses to expect.
- **Testing and Debugging:** The "Try it out" feature in Swagger UI is an invaluable tool for testing your endpoints during development.
- **Code Quality:** The requirement to use type hints and Pydantic models for automatic documentation encourages writing cleaner, more explicit code.

You've now seen your API come to life with a user-friendly interactive interface, a testament to FastAPI's focus on developer experience.

Next Steps and Further Learning

Congratulations! You've successfully built a functional REST API using FastAPI, covering essential concepts like environment setup, route definition, data validation with Pydantic, dependency injection, basic error handling, and testing. You've also explored the powerful automatic interactive documentation.

This is just the beginning of what you can do with FastAPI. Here are some concrete ideas to extend this project and deepen your understanding:

What to Build Next

1. **Integrate a Database:** Our current API uses a simple in-memory dictionary, which resets every time the server restarts.
    - **Challenge:** Replace `in_memory_items` with a persistent database.
    - **Ideas:**
        - **SQLite with SQLAlchemy:** A lightweight option for local development. Use SQLAlchemy ORM to define models and interact with the database.
        - **PostgreSQL with SQLAlchemy/SQLModel:** For a more robust solution, connect to a PostgreSQL database. Consider using SQLModel, a library built by the creator of FastAPI, which combines Pydantic and SQLAlchemy for a seamless experience.
    - **Skills you'll learn:** Database connection management, ORM usage, data migration, persistent storage.
2. **Add User Authentication and Authorization:** Most real-world APIs need to secure their endpoints, allowing only authenticated and authorized users to access certain resources.
    - **Challenge:** Implement user registration, login, and secure endpoints.
    - **Ideas:**
        - **JWT (JSON Web Tokens):** Use `python-jose` and FastAPI's `Security` and `Depends` to implement token-based authentication.
        - **OAuth2:** FastAPI has built-in support for OAuth2, which is the industry standard for authentication and authorization.
    - **Skills you'll learn:** Hashing passwords, token generation and validation, security dependencies, role-based access control.

3. **Deploy Your API:** Get your API out of your local development environment and onto a server where others can access it.
    - **Challenge:** Deploy your FastAPI application to a cloud provider.
    - **Ideas:**
        - **Docker:** Containerize your application using Docker. This makes it portable and ensures it runs consistently across different environments.
        - **Cloud Platforms:** Deploy to platforms like Heroku, AWS EC2/ECS, Google Cloud Run, or Render. These platforms offer various ways to host Python web applications.
    - **Skills you'll learn:** Dockerfile creation, containerization, cloud deployment strategies, environment variables for configuration.

Further Learning

The official FastAPI documentation is an excellent resource, renowned for its clarity and comprehensive examples. It covers everything from advanced dependency injection to WebSockets, background tasks, and much more. Keep exploring, keep building, and enjoy the power of FastAPI!