Session II

Improving and Containerizing the Web API

Web API Advancements

Outline:

- Input validation
- Error handling
- Logging
- Documentation
- Testing

Input Validation

Prevent invalid and malicious input from causing errors or unexpected behavior.

Key Aspects:

- Data types: ensure expected types (e.g., int, str)
- Required fields: prevent missing values
- Format: check lengths, ranges, formats, predefined choices
- Logical consistency: alignment of related fields
- Clear response: meaningful messages on validation errors

Input Validation in FastAPI

Task:

- Try to send invalid requests to your API,
 e.g., http POST http://127.0.0.1:8000/chat max_length=test
- Inspect the responses
- Try to send requests that are valid but cause an Internal Server Error

Input Validation in FastAPI

Pydantic for input validation:

```
from typing import Optional
from pydantic import BaseModel, Field

class CreateBookRequest(BaseModel):
   title: str = Field(..., min_length=1, max_length=100)
   author: str = Field(..., min_length=1, max_length=50)
   year: int = Field(..., ge=1000, le=2100)
   pages: int = Field(..., gt=0)
   genre: Optional[str] = Field(None, max_length=30)
```

FastAPI validation error response:

Input Validation in FastAPI

Validation functions for more complex input validation:

```
class CreateBookValidRequest(BaseModel):
    ...
    @field_validator("year")
    @classmethod
    def validate_even_year(cls, year):
        """Ensure that the year is an even number."""
        if year % 2 != 0:
            raise ValueError("Year must be an even number")
        return year

@model_validator(mode="after")
    def validate_science_fiction_year(self):
        """Ensure that Science Fiction books are from 1950 or later."""
        if self.genre and self.genre.lower() == "science fiction" and self.year < 1950:
            raise ValueError("Science Fiction books must be published in 1950 or later")
        return self</pre>
```

Error Handling

Handle errors gracefully to ensure reliability and clear communication.

Key Aspects:

- Catch exceptions: prevent crashes with proper error handling
- Categorize errors: distinguish between client (4xx) and server (5xx) issues
- Provide useful responses: return clear, actionable error messages
- Fail safely: avoid exposing sensitive data or breaking functionality
- Log errors: capture details for debugging and monitoring (next step)

Error Handling

Common Response Codes:

- 200 OK Request successful
- 201 Created Resource successfully created
- 400 Bad Request Invalid input or malformed request
- 401 Unauthorized Authentication required
- 403 Forbidden Access denied
- 404 Not Found Resource does not exist
- 422 Unprocessable Entity Valid request, but input violates business rules
- 500 Internal Server Error Unexpected server issue

Error Handling in FastAPI

HTTPException:

```
@app.get("/books/{book_id}")
def get_book(book_id: int):
    if book_id not in books_db:
        raise HTTPException(status_code=404, detail="Book not found")
    return {"book_id": book_id, "title": books_db[book_id]}
```

Response:

```
HTTP/1.1 404 Not Found
{
    "detail": "Book not found"
}
```

Exception Handler: define response by type

• e.g., *Exception* for unknown errors

```
class BookNotFoundException(Exception):
    def __init__(self, book_id: int):
        self.book_id = book_id

@app.exception_handler(BookNotFoundException)
async def book_not_found_handler(req: Request, exc: BookNotFoundException):
    return JSONResponse(
        status_code=404,
        content={"error": "Book not found", "book_id": exc.book_id},
    )

@app.get("/books/custom/{book_id}")
def get_custom_book(book_id: int):
    if book_id not in books_db:
        raise BookNotFoundException(book_id)
    return {"book_id": book_id, "title": books_db[book_id]}
```

Logging

Key Aspects:

- Capture errors: log exceptions for debugging
- Use appropriate levels: INFO, WARNING, ERROR, CRITICAL
- Avoid sensitive data: never log user credentials or personal info

Python logging module: import logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

logger.warning("This is a warning")
logger.error("Oops! An error")

Documentation

OpenAPI Specification:

- Standardized contract: defines endpoints, parameters, and responses
- Machine-readable format: supports tools like Swagger & Postman
- Auto-generation by FastAPI

```
/books/{book_id}:
   get:
    summary: "Retrieve a book"
    description: "Fetch book details by ID."
    parameters:
        - name: "book_id"
        in: "path"
        required: true
        ...
```

Documentation in FastAPI

Improved automatic documentation with metadata, descriptions and examples:

Metadata:

app = FastAPI(title="Library API", description="API for managing...", version="1.0", contact={ "name": "API Support", "email": "support@example.com", }, license info={

Fields:

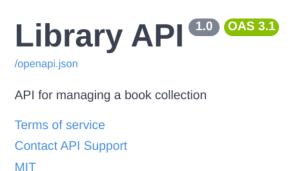
```
title: str = Field(
    ..., example="1984", description="Book title"
)
year: int = Field(
    ..., example=1949, description="Publication year"
)
```

Endpoints:

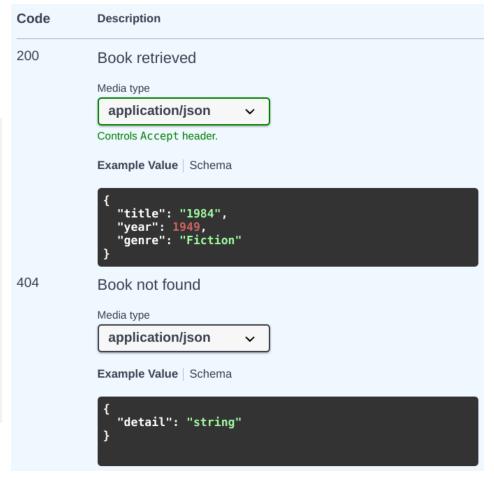
```
@app.get(
   "/books/{book_id}",
   response_model=Book,
   summary="Retrieve a book",
   tags=["Books"],
   responses={
      200: {"description": "Book retrieved"},
      404: {
         "description": "Book not found",
          "model": ErrorResponse
      },
   },
```

Documentation in FastAPI

Improved documentation:







Practice: Improve Web API

Task: Improve your API implementation by adding

- Input validation (remember your requests from earlier)
- Error handling for possible and unexpected errors
- Logging for requests and errors
- Improved documentation (see Swagger UI for the changes)

Test your application using the UI and/or the console

Feel free to expand the functionality if time allows, e.g., add configurations for inference

Hint: Revisit slides, look at the examples folder and search the web

Testing

Testing a web API:

- Status codes: verify correct responses (200, 404, 400, ...)
- Response data: check JSON structure, types, values
- Input validation: check validation rules
- Error handling: check structured error responses
- Edge cases: test empty, large, and unexpected inputs (e.g., special symbols)
- And more: authentication and permissions, headers, rate limits, high loads

Testing FastAPI

Testing FastAPI:

```
from fastapi.testclient import TestClient
from fastapi_error_handling import app # Import your FastAPI app

client = TestClient(app)

def test_get_book_success():
    response = client.get("/books/1")
    assert response.status_code == 200
    assert response.json() == {"book_id": 1, "title": "1984"}

def test_get_book_not_found():
    response = client.get("/books/999")
    assert response.status_code == 404
    assert response.json() == {"detail": "Book not found"}
```

- Run tests with pytest my_tests.py
- For POST: client.post("/", json={...})

Task: Create tests for your endpoint

 Test various aspects of your API, input validation, error handling, proper responses on both error and success, ...

Containerization

Containers package applications with all dependencies for consistent execution

Key Benefits:

- Portability: run the same image anywhere (local, cloud, server)
- Isolation: avoid conflicts with system dependencies
- Scalability: easily deploy multiple instances
- Reproducibility: ensures the app behaves the same across environments

Docker Basics

Docker Terminology:

- Image: packaged application with everything needed to run (code, dependencies, OS libraries)
- Container: running instance of an image, isolated from the system
- Dockerfile: script that defines how to build an image
- Registry: repository for storing and sharing images (e.g., Docker Hub)
- Volumes: persistent storage for data across container restarts
- Networks: allow communication between containers

How to Docker

Steps:

- Write Dockerfile: define base image, files, dependencies, starting command, ...
- Build an image: docker build -t myapp:v1.
- Run a container: docker run -p 8000:8000 myapp:v1
- Share via registry: docker tag myapp:v1 myrepo/myapp:v1 docker push myrepo/myapp:v1
- Pull an image: docker pull myrepo/myapp:v1

FastAPI Dockerfile (with uv)

```
# Define base image.
       Base image FROM python: 3.12-slim
                      # Install uv.
                      COPY --from=ghcr.io/astral-sh/uv:latest /uv /uvx /bin/
                      # Copy the application into the container.
         Copy files COPY . /app
                      # Install the application dependencies.
              Install WORKDIR /app
                      RUN uv sync --frozen --no-cache
    dependencies
                      # Indicate the port for correct usage.
                      EXPOSE 80
                      # Run the application.
Starting command CMD ["/app/.venv/bin/fastapi", "run", "app/main.py", "--port", "80", "--host", "0.0.0.0"]
```

Practice: Dockerize your API

Task:

- Build and run your API as a Docker container
- docker build -t inference-api .
- docker run -p 127.0.0.1:8000:80 -d --name inference-api inference-api
- Try to send requests as before
- List containers: docker ps
- Show logs: docker logs inference-api
- Stop container: docker stop inference-api