Resmed Assignment:

Author: Mauro De Luca

User Story:

As a university professor, I want a simple way to share resource files with my students.

Intended users: Professor and students

Requirement:

- Build a lightweight file-sharing API
- Supports uploads storing files on disk
- Makes it easy to retrieve shared files via a simple HTTP endpoint (downloads)

Simple:
- User-friendly
- Easy to use
- No unnecessary feature (follow requirements)

Lightweight:
- Not resource heavy
- Fast and efficient
- Doesn’t have a lot of dependencies and overheads

Design:

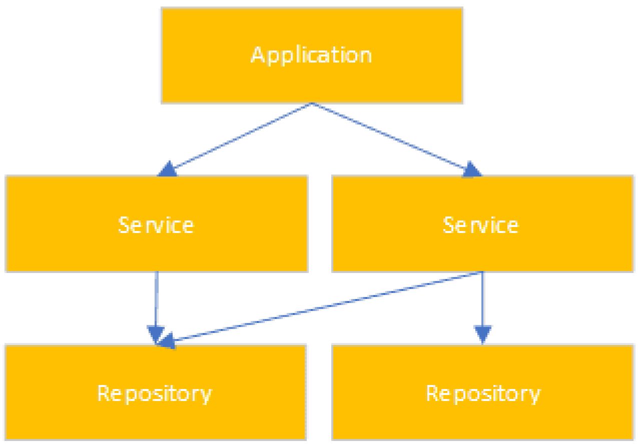
Framework:

I have chosen to use the Python FastAPI web framework since it is fast to code, easy to learn and high performance. Some advantages of using FastAPI are:

- Robust and intuitive
- Documentation and validation are automatic
- Uses modern Python (type hints, async, etc)
- Faster than Flask
- Removes a lot of boilerplate

Architecture:

When I first began to develop the application, most of the logic was contained in the routes. This worked well however it made the architecture rigid and difficult to test. As I started to refactor and make incremental improvements to the code, I decided that it was better practice to develop in a more modular way, even if it meant losing some minimality. When I say “modular” I mean if a particular functionality were to be updated (such as replacing the central json with a NoSQL database), it would be easier to replace without disrupting the rest of the application. So, this makes the application more maintainable.



In my final design, the application is architected according to the controller-service-repository pattern.

There are many benefits in following this structure. This layered approach allows for a separation of concerns in the application, by giving each layer of abstraction different responsibilities and purposes.

- The controllers or routes handle HTTP requests/response and manage the REST interface to the services.
- The services implement the business logic (validations, orchestrations, decisions).
- The repository handles data persistence (any IO operations and transactions)

My application has three routes:

- Files: handles files API
- Course: hosts the /course web page
- Professor: hosts the /professor web page

The files service handles file operations and metadata management. It is injected into the system via dependency injection and is by default transient (a new instance of the service is created every time it is requested). I designed the service to be independent of the FastAPI framework, which improves portability.

There are two repositories, file and metadata. The file repository is responsible for storing and retrieving the file on the disk while the metadata repository manages the metadata of all the files.

Error handling is managed through the handlers if the exception is known or by the middleware if the exception is unexpected.

In conclusion, even though I've added more layers, there is a more logical flow to the applications. Overall, the code becomes more readable and future proof.

Endpoints:

Endpoints should represent resources, not actions.

- POST /files → upload
- GET /files → list
- GET /files/{id} → download

ID vs Filename

Using the original filename to store the file is not the best option in terms of functionality. If the professor uploads a file with the same name (even if the file has different content), there will be a collision on the disk. The data that was stored under that file path will be lost/overwritten.

Storing the file with the ID is the better option. The ID and associated name can be tracked in the metadata json. The friendly name for the file can be returned to the user in the download response via the Content-Disposition filename parameter.

Metadata

I have decided to implement a one central json file to store all the metadata for the files.

Central JSON File:

The advantages are it's simple, easier to list files, good enough for a classroom-scale project. It also makes listing all the files' metadata a quick operation since the contents of metadata json just has to be dumped to the user.

The disadvantages are if the file gets corrupted, metadata for all files can be lost. There could also be an issue with concurrency if multiple operations are occurring on the file at once. I shall implement atomic operations to avoid race conditions. Querying and filtering is manual and inefficient for large datasets - however for this demo I assume the dataset won't scale massively.

Multiple JSON Files:

A metadata file could be created for each file and stored under the file ID. This reduces the risk of losing all the data from corruption. It also would make listing the files API slower since the directory would have to be scanned to collect all the metadata for each file.

NoSQL Database:

A NoSQL database, such as NoSQL Lite, MongoDB etc, would be the best implementation if the project were to a larger size. It ensures scalability, flexibility and safety. However, it adds external dependencies and a lot more setup. I assume this service is small scale

Model:

I decided to not create a model for the entry to the file metadata json but instead I have type checks in the service and repository layer.

A model could be added later with Pydantic.

Resources:

I followed the FastAPI official tutorial to develop the code.
I followed a service-repository pattern.

Assumptions:

- Service shall serve a class of < 30 students.
- Service shall serve one professor who shall upload < 100 files.
- Only professor shall be able to upload a file of MAX size 20MB using a /upload endpoint
- Students and professor can call the GETs
- Students are all in the same class.
- All students can access files
- Professor can reupload a file with a new timestamp and a history of those files shall be listed
- Students do not have access to /professor. Authentication was not a requirement so I just obscured the API, which obviously isn't safe IRL but for demo purposes let's assume it's ok.
- The table data doesn't need to be sorted
- All students are part of one course

Instructions:

All steps to run and test the application are included in the README.md file in the root directory of the project.

Questions:

What happens if something goes wrong during a request?
When something goes wrong during a request to my application, the error is handled according to the point of failure and an appropriate response is returned to the endpoint.

FastAPI handles errors relating to request and response validation automatically. 400 errors are errors due to the user while 500 errors are related to the server. If a known error occurs on the server side due to the user misusing the system, the exception is caught by the appropriate exception handler. Whereas, if the error is not known then it's caught by the middleware generic exception handler.

How does the API communicate this to a client?
The API communicates with appropriate HTTP error codes and "detail" messages. The errors can then be displayed to the user on the webpages if something goes wrong.
On the professor page, the error messages show in red if the upload fails. For example, if the user exceeds the MB limit, a message will inform the user that the file they are trying to upload was not accepted.
On the student page, if something goes wrong with the listing of the files metadata the message "Failed to load files" is displayed. The details of the error are output on the browser's console.

How can you confirm the code works?
When you say the code works, I assume you mean it does what is intended according to the user's needs/requirements.

How can someone else run and test the API quickly?
An end user who doesn't have the technical knowledge could test my API via the webpages, provided they were hosted for them somewhere.

A developer can run the tests (pytest), which checks cases where the application should work and not work. Running the test is quicker because the server doesn't have to be running. The APIs are mocked.

There is also API documentation available at /docs for Swagger UI or /redocs for Redocs, which a developer can use to test each API.