Our project is an online bookstore using a multi-tier (multi-service) architecture. Instead of having all the code in one big application, we broke the system into smaller chunks, each performing one task. This type of architecture is often called a microservices architecture.

We created three main pieces (servers):

Gateway Server – This is the entrance. All user requests go through here first. It directs the request to the correct service.

Catalog Server – This handles all things about books: searching, listing by topic, and presenting book details.

Order Server – This part takes care of purchasing books and checking if a book is available or not.

Both parts run separately in their own container with the assistance of Docker, making it convenient for us to handle and run everything.

Assume a user wants to search for and buy a book:

The user sends a request to the Gateway Server.

If they want a book, the Gateway asks the Catalog Server, which gets books by subject.

If the user chooses a book to get more details, the Catalog Server provides details like title, price, and availability.

To buy a book, the Gateway asks the Order Server.

The Order Server checks if the book is available (asks the Catalog Server).

If the book is available, stock is updated and purchase is made.

All this is being done through HTTP requests between the servers using REST APIs.

Microservices vs One Big Program (Monolith)

We chose to split the program into services so that each service is small and specialized in one task. It is simpler to fix or upgrade without ruining other parts.

But, using multiple services also means more communication between parts, which is slower or makes errors more difficult to trace.

Simple REST API

We used plain HTTP requests (GET, POST) that are easy to learn and debug.

In complex systems, people use message queues or more sophisticated communication, but for learning purposes, we simplified it.

Data Storage

Now, we do not use a database. Information for all books is stored in plain arrays (in memory).

It's fine for testing purposes, but information gets lost with every server reboot. A database would be more suitable for an actual version.

If you reboot a server, all book and order data are lost (no database used).

No account system or user login, so everyone can put requests in.

No error messages are shown to the user in a friendly way. When something does break, it just shows raw errors.

Gateway Server is a single point of failure — when it crashes, the entire thing crashes.

Some things we considered but didn't do (yet):

Use an actual database (like MySQL or MongoDB) to store book and order information permanently.

Implement user accounts, so users can register and see their order history.

Implement a frontend website using HTML/React, so users can click and buy books in a browser instead of curl or Postman.

Implement error handling and nice messages.

Implement a load balancer or backup gateway server, so the system does not go down if one piece of hardware crashes.

Implement tokens or authentication, so only real users can buy books.