Module 4 — Backend Server

Lecture 1: Spring Framework and Spring Boot

Spring core ideas

IoC and DI

Why Spring Boot

Spring Boot vocabulary

Objectives

What you’ll learn today?

- Explore the purpose of a backend server using a request-response example
- Define dependency, dependency injection, and inversion of control using
- Explain why Spring Boot exists and it’s benefits over plain Spring setup
- Understand a request-response cycle
- Build a small Student mini-project and identify where wiring happens

Today's flow

- Backend server mental model
- The pain point in Java
- Spring: core ideas
- Layers used in backend projects
- Guided build: Student mini-project
- Spring Boot: why it exists and key terms
- Spring Boot: startup and annotations preview
- Recap

Transition: From modules 1-3 to module 4

You already know

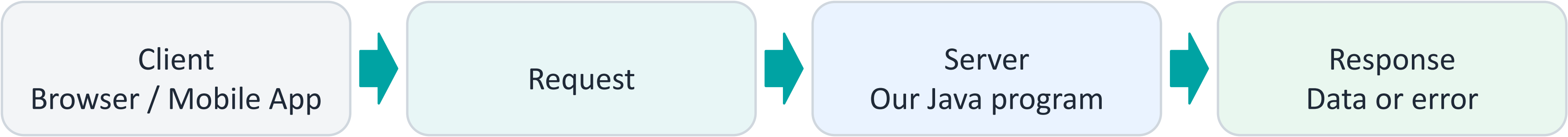
- Core Java
- Writing logic with DSA
- Working with data in databases

Now we add

- Backend server mindset
- API request and response
- Project structure used in backend code
- Spring to manage object creation and wiring

Basics: Backend Server

What is a backend server?



A server is a program that stays running, listens for requests, runs code, and sends back responses.

In this module, we will turn Java logic into something other applications can call.

- Question: Pick any app you use daily. What is one request it might send, and what response it might get back? Answer in chat.
- Key takeaway: A request asks for something. A response brings back data or an error.

Request, Response, Endpoint

- **Endpoint** is a specific URL path on the server; for example, *GET /students*
- **Request** is a message sent to the server asking for something; for example, *give me all students*
- **Response** is a message sent back with the result; for example, *a list of students or an error*

- Question: If the endpoint is GET /books, what do you expect back? Answer in chat.
- Key takeaway: A list of books, usually in JSON.

Why Spring Exists?

The problem we are solving in Java

- As applications grow, object creation starts spreading everywhere with *new keyword*
- Changes begin to touch business logic even when the business rule did not change
    - ○ Business logic: A plain-language decision or policy the product wants. Example: “When a student registers, save the student and send a welcome message.”
    - ○ Business rule: The actual code that implements those rules. Example: the registerStudent() method validates input, saves the student to a database and sends an email notification
    - ○ Consider: the code change when deciding to store students in MySQL database instead of text files.
- Testing becomes harder because dependencies cannot be replaced easily
- This results in a tightly coupled application

- Question: Where do you usually create and connect objects in a plain Java app today? Answer in chat.
- Key takeaway: Usually in main or bootstrap code, and it grows as the app grows.

What’s tightly coupled?

```
public class StudentService {
  private final StudentRepo repo = new MySqlStudentRepo();
  private final Notifier notifier = new EmailNotifier();
}
```

- Classes that are highly dependent on each other are **tightly coupled**; so, one class directly creates, controls, or relies on the specific implementation of another
- Example change requests that force edits in StudentService:
    - Switch storage from MySQL repository to MongoDB repository
    - Switch notifications from Email to WhatsApp or SMS
    - Use a FakeNotifier class for testing so no real messages are sent to students

What is Spring Framework?

- Spring Framework is a Java framework that helps you build large applications by managing object creation and dependency wiring for you
- It provides a container that creates objects, connects them together, and manages their lifecycle
- The main value: your business code focuses on “what to do,” while Spring handles “how objects are created and connected”
- Library vs Framework:
    - With a library, your code calls the library when you need it
    - With a framework, the framework calls your code and controls the overall flow (you plug your code into its structure)

Dependency Injection & Inversion of Control

What’s a dependency?

- A dependency is something a class needs to do its job
- Examples
  - A StudentService class needs a repository to store and fetch students
  - StudentService needs a notifier to send a message

- Question: In the previous code, what are the dependencies of StudentService? Answer in chat.
- Key takeaway: StudentRepo and Notifier are dependencies.

What’s dependency injection (DI)?

Dependency injection means a class does not create its dependencies. It receives them from the outside.

Before

```
public class StudentService {
  private Notifier notifier = new
EmailNotifier();
}
```

After

```
public class StudentService {
  private final Notifier notifier;

  public StudentService(Notifier notifier) {
    this.notifier = notifier;
  }
}
```

In the *After* scenario, Notifier can be an interface (ideal) or a parent class

- Question: In the after version, who decides which Notifier we use? Answer in chat.
- Key takeaway: Something outside the class decides, for now App or main; later Spring.

How does dependency injection help?

- If requirements change, we can swap implementations without editing business logic
    - Example: Today we send notifications via email. Later, they may need to be sent to WhatsApp
    - So, we’ll simply create a WhatsAppNotifier class that implements the Notifier interface.
- Testing becomes easier because we can replace real dependencies with fake ones
- We can start with in-memory or file storage and move to a database later with less rewriting
    - Example: At first we store students in memory so the app runs quickly without any setup. Later we switch to database storage so data is not lost when the program restarts

What’s inversion of control (IoC)?

- Inversion of control means your code is not responsible for creating and managing objects; a container takes that control
- In Core Java: your code creates objects and wires them
- In Spring: the container creates objects and wires them

DI: without vs with Spring

- In plain Java, you can pass dependencies through the constructor, but you still have to decide and create them in one place (usually main)
- As the project grows, that “wiring” code becomes a messy setup area you keep editing whenever a dependency changes
- With Spring, you mark which classes should be managed, and Spring creates the objects and connects them automatically
- Result: when you swap a dependency, you usually change configuration or one implementation choice, not business logic across multiple files

Mini-Project: Structure & Implementation

Structure: Controller, Service, Repository

- **Controller** receives a request and returns a response, and stays thin and focused on input and output
- **Service** contains business rules and decisions, and it is where the core logic lives
- **Repository** stores and fetches data, in memory today and database later
- Real systems will have additional layers
  - Example: for dependencies on external collaborators like email or SMS; we use a **notifier** to make dependencies realistic
- Why’re we covering this?
  - Our Java project structure will evolve to better organize our code using these concepts

- Question: If a registering student should be sent a welcome message, which layer owns that rule? Answer in chat.
- Key takeaway: Service owns the rule; controller only triggers it.

Your turn!

Decide where each line belongs (service, controller or repository):

- Validate email format before registering
  - Service
- Return HTTP 400 if input is missing
  - Controller
- Store student details
  - Repository
- Decide whether to notify the student
  - Service

Today's mini-project

- I’ll share a link to the skeleton & you’ll implement!
- Objectives
  - Register a student with name and email
  - List all students
  - Send a welcome notification after registration
  - Start with in-memory storage for all students
- Not covered today
  - No database integration in today
  - No Spring Security or authentication
  - No JPA or Hibernate for database interaction
  - No Maven, pom.xml, or application.properties

Implementation

- Tasks
  - Create Student model (id, name, email)
  - Define StudentRepository interface (save, findById, findAll)
  - Implement InMemoryStudentRepository (use Map<UUID, Student>)
  - Define Notifier interface (send)
  - Implement ConsoleNotifier (print readable messages)
  - Build StudentService with constructor injection (registerStudent, listStudents)
  - Build AttendanceService with constructor injection (markPresent)
  - Wire everything in App (new repo/notifier/services) and run a demo flow
- Definition of Done
  - A student is registered
  - A welcome notification is printed
  - Listing students shows the registered student

- Question: What output would convince you the notifier dependency is working? Answer in chat.
- Key takeaway: You see a welcome message printed for the registered student.

Spring Boot

Why Spring Boot exists?

- Spring helps with object management and wiring at scale
- Spring Boot exists because starting a Spring application used to require lots of setup
- Spring Boot helps you start a backend application quickly
- What does Spring Boot give us?
    - A runnable backend app with the right wiring already set up to start cleanly
    - An embedded server so your Java program can accept HTTP requests immediately
    - Auto-configuration that connects common pieces without you writing setup code first

- Question: What does embedded server mean in one line? Answer in chat.
- Key takeaway: The server runs inside your application process.

Defaults in Spring Boot

- Defaults mean Spring Boot makes common choices so you do not write boilerplate setup code upfront
- If you add *spring-boot-starter-web* as a dependency, Boot starts an HTTP server so your app is reachable at http://localhost:8080 from your browser
- You do not write code to register controllers; Boot enables annotations like @RestController
    - We'll cover annotations in the upcoming section!
- When we later add a database dependency, Boot can create connection-related setup from configuration instead of a custom DB connection manager class

- Question: Which of these would be the biggest time saver in a new project? Answer in chat.
- Key takeaway: You avoid setup glue code and focus earlier on business logic.

Spring Framework and Spring Boot

- Spring Framework is the container that creates objects and injects dependencies
- Spring Boot is the starter kit that makes a Spring app runnable quickly as a backend server
- Spring Boot does not replace Spring; it uses Spring and adds conventions so you write less setup code

Spring Boot: A Startup Story

- It starts the application entry point (similar to “main” in our program)
- It scans your code to find components
- It creates beans for those components
- It injects dependencies by calling constructors
- It starts the embedded server so it can accept requests

- Question: When the server is running, what happens when a client calls GET /students? Answer in chat.
- Key takeaway: The request reaches a controller method, which calls service logic and returns a response.

Spring Boot: Vocabulary

What’s a Bean?

- A bean is an object that Spring creates, stores, and manages for you
- Because Spring manages it, Spring can reuse it, inject it into other classes, and control its lifecycle
- In our Student example, what’re the objects we create?
    - We create objects for repository, service, and controller; these are good candidates for beans

What’s a Component?

- A component is a class Spring can discover and manage
- You mark it using annotations so Spring knows it should create it
- In our Student example, what’re the *components*?
  - StudentController as a controller component
  - StudentService as a service component, and
  - InMemoryStudentRepository as a repository component

- Question: If Spring does not discover a class, what can it not do for that class? Answer in chat.
- Key takeaway: It cannot create it as a bean, so it cannot inject it into other classes.

Dependency in Spring terms

- A dependency is still the same idea: one object needed by another
- In Spring, constructor parameters are treated as dependencies
- Spring uses the constructor to inject the right beans

- Question: If StudentService constructor takes StudentRepository, what is the dependency? Answer in chat.
- Key takeaway: StudentRepository is a dependency of StudentService.

Requests and controllers

- A request is what a client sends to your server; for example: the client calls GET /students
- The request arrives at a controller method; for example: StudentController.students()
- The controller calls the service; for example: studentService.getAllStudents()
- The controller returns the response; for example: a list of students as JSON

- Question: Why should a controller avoid business rules? Answer in chat.
- Key takeaway: Business rules belong in services so they can be reused and tested without HTTP.

Where’s dependency injection happens in startup?

- Boot scans the application to find components
- It creates beans for those components
- It injects dependencies by calling constructors with the required beans
- After wiring is complete, the server starts and requests can be handled

Annotations

Refresher: Java annotations

- An annotation is metadata attached to a class or method
- It adds meaning without changing the business logic of the method
- Spring reads annotations to decide what to create and how to connect it

Spring Annotations: An Intro

- **@SpringBootApplication** marks the entry point and starts Spring Boot
- **@RestController** marks a class that handles HTTP requests
- **@Service** marks business logic classes
- **@Repository** marks data access classes

Practice: Annotation

- **Given our classes**: StudentController, StudentService, StudentRepository
- **Question**: Which annotation would you expect on each class? Answer in chat.
- **Key takeaway**: StudentController: RestController; StudentService: Service; repository: Repository.

Mapping the our project to Spring Boot

- In our project, App.java did manual wiring using *new keyword*
- In Spring Boot, the container does the wiring
- Our service and repository logic can stay almost the same; what changes is how objects are created and connected
- Controllers become the entry point for requests instead of main

- Question: What part of our app will Spring Boot replace first? Answer in chat.
- Key takeaway: The manual wiring in App or main.

Next step preview

- Expose /students as an HTTP endpoint
- Let Spring create and inject the service and repository
- Run the app and test using browser or curl

- Question: What is the single biggest difference between our app and a server? Answer in chat.
- Key takeaway: our app runs once; server stays running and responds to many requests.