🟢 Case Study Title: *Online Course Enrollment System*

🧩 Scenario:

An educational startup wants to build a basic web application for students to view available courses and enroll online. The company has a small IT team familiar with Java and wants to use **Spring MVC** to ensure the application follows a clean, maintainable structure based on MVC architecture.

🎯 Objectives:

1. Display a list of available courses.
2. Allow students to register by filling out an enrollment form.
3. Confirm enrollment and store student details.

🏗️ System Requirements:

- Java 17 or later
- Spring MVC framework
- Apache Tomcat or embedded server
- Maven for dependency management
- JSP for frontend
- Eclipse or Spring Tool Suite (STS) IDE

🧠 How Spring MVC Helps:

Spring MVC allows the application to be divided into three main components:

Layer	Responsibility
Model	Represents the data (Course, Student, Enrollment info)
View	Displays the HTML pages for course listing and form input
Controller	Manages user requests and application logic

🔁 Application Flow:

1. **User accesses the homepage**
   → A controller handles this request and returns a list of available courses via the view.

2. **User selects a course and proceeds to enroll**
   → A new view (HTML form) is presented to collect user data (name, email, etc.).

3. **Form is submitted**
   → The controller receives the form data, validates it, and passes it to the service layer or model to be processed.

4. **Success page is shown**
   → A confirmation view is displayed with enrollment details.

🧩 Components in Spring MVC:

Component	Description
`@Controller`	Handles web requests (e.g., show courses, process enrollment)
`@RequestMapping`	Maps URLs to specific controller methods
`Model` object	Holds the data to be passed to the view
`@ComponentScan`	Auto-detects components (controllers, services, etc.)
`ViewResolver`	Resolves the view name to an actual view (e.g., JSP page)
`Beans.xml` or Java Config	Defines Spring beans, view resolvers, and component scanning setup

🧾 Example Use Cases:

1. **CourseController**
   - `/courses` → Displays list of courses
   - `/enroll` → Shows enrollment form
   - `/submitEnrollment` → Processes submitted data
2. **Views (JSP)**
   - `courses.jsp` → Displays all courses
   - `enroll.jsp` → Input form for registration
   - `success.jsp` → Confirmation message

✅ Case Study Title: *Online Shopping Portal – Order Processing Monitoring*

🧾 Scenario Description

An **online shopping portal** provides a service class `OrderService` that has three key methods:

1. `addToCart(String product)`
2. `placeOrder(String orderId)`
3. `cancelOrder(String orderId)`

As a developer, you want to add **cross-cutting concerns** like:

- Logging when methods start (`@Before`)
- Logging after successful method execution (`@AfterReturning`)
- Logging errors when a method fails (`@AfterThrowing`)
- Performing cleanup or logging after any method execution, success or failure (`@After`)

📦 Spring AOP Setup Components

1. Business Logic Class

`OrderService` — contains methods like `addToCart`, `placeOrder`, `cancelOrder`.

2. Aspect Class: `OrderLoggingAspect`

This class uses four annotations:

Annotation	Purpose
`@Before`	Logs method entry
`@AfterReturning`	Logs method success result
`@AfterThrowing`	Logs if any exception occurs
`@After`	Logs method exit regardless of outcome

🔄 Flow with Annotations

Let’s walk through what happens when a user places an order.

✅ Method: `placeOrder("ORD123")`

Step	Annotation	What Happens
1	`@Before`	Log: “Starting method: placeOrder with order ID: ORD123”
2	— Business Logic —	The order is placed successfully
3	`@AfterReturning`	Log: “Order placed successfully: ORD123”
4	`@After`	Log: “Method placeOrder execution finished”

❌ Method: `placeOrder("INVALID_ID")`

Step	Annotation	What Happens
1	`@Before`	Log: “Starting method: placeOrder with order ID: INVALID_ID”
2	— Business Logic —	Throws exception: OrderNotFoundException
3	`@AfterThrowing`	Log: “Exception while placing order: OrderNotFoundException”
4	`@After`	Log: “Method placeOrder execution finished”

📌 Aspect Class Summary

Advice Type	Trigger Condition	Example Log Message
`@Before`	Just before the method execution	"Calling method: addToCart"
`@AfterReturning`	When method returns successfully	"addToCart completed successfully for product: X"
`@AfterThrowing`	When method throws an exception	"Error occurred during addToCart: ProductNotFound"
`@After`	After method finishes (success or error)	"addToCart method execution ended"