Enterprise Programming
(Course Slides)

Module2

Refer for detailed notes :-

https://github.com/sagaruppuluri/EP/blob/main/Module2/Readme.md

Modular Architecture

- **Separation of Concerns or Single Responsibility –** Each module should have single responsibility.
- **Encapsulation –** Modules should hide their internal implementation and expose only necessary interfaces.
- **Loose/Low Coupling -** Modules should interact with each other through well-defined interfaces, minimizing dependencies.
  - Low Coupling => Low Dependence => Hence DESIRABLE
- **High Cohesion -** Related functionalities should be grouped together.

Clean Code Practices

- **Meaningful Names** – Use descriptive and meaningful names.
- **Single Responsibility Principle** – One responsibility for a class.
- **Avoid Magic Numbers** - Use named constants instead of hardcoded numbers.
- **Write Tests** - Implement unit tests and integration tests to ensure correctness.
- **Dependency Injection** - loosely couple code using DI.
- **Small Functions** - Functions should be small.
- **Document Code** – Use comments and documentation.
- **Don’t repeat yourself (DRY)** - avoid code repetitions

Spring vs Spring Boot

Feature	Spring Framework	Spring Boot
Configuration	Complex XML/Java config	Auto-configuration
Setup Time	Hours/Days	Minutes
Server	External (Tomcat)	Embedded
Dependencies	Manual management	Starter POMs
Best For	Large enterprise apps	Microservices, REST APIs

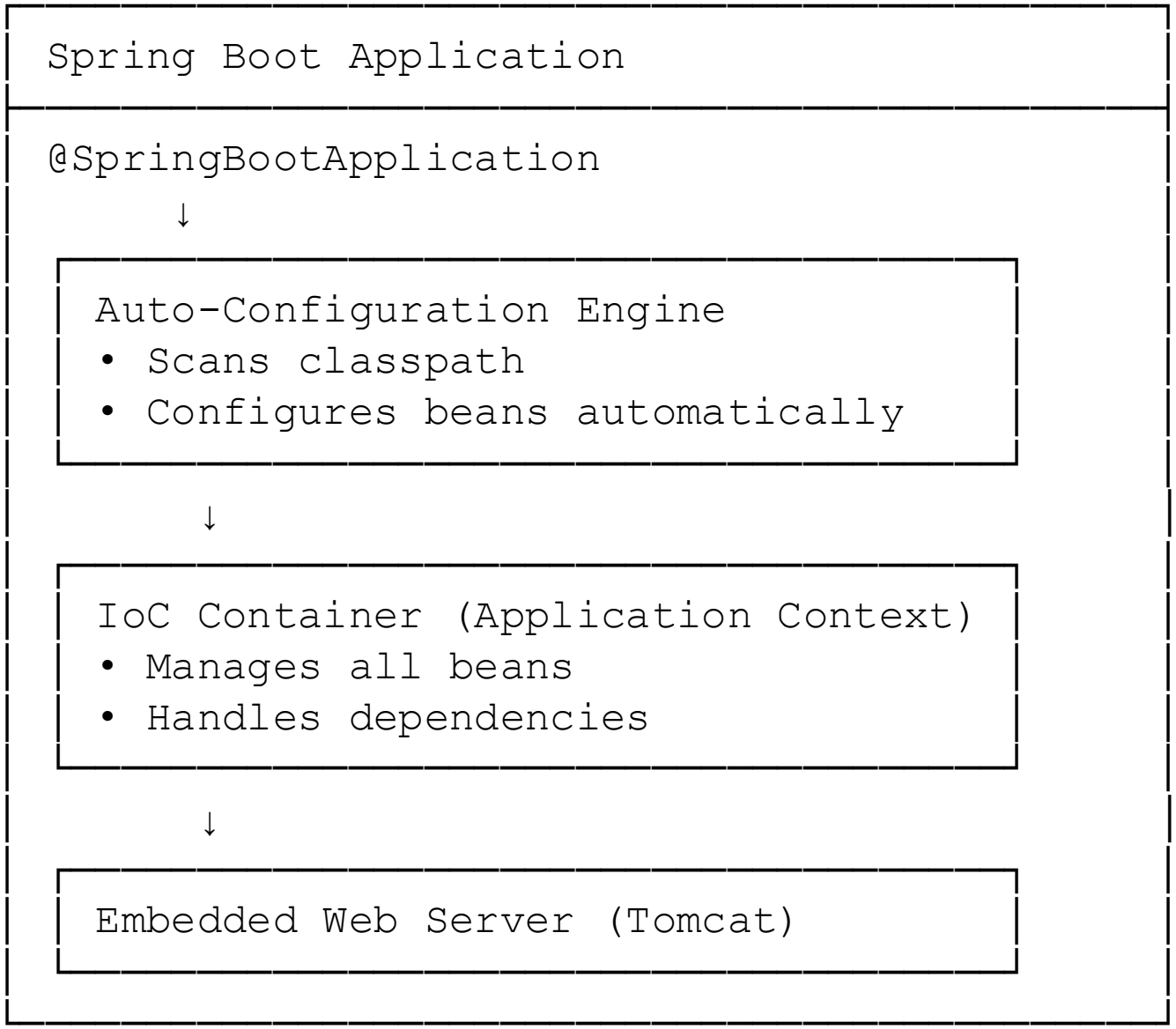
Spring Boot - Introduction

Spring Boot is a framework that makes it easy to create stand-alone, production-ready Spring applications.

Key Features:

- ✅ **Auto-Configuration** - Automatically configures your application
- ✅ **Embedded Server** - No need for external Tomcat/Jetty
- ✅ **Production-Ready** - Metrics, health checks built-in
- ✅ **No XML Configuration** - Use annotations instead
- ✅ **Starter Dependencies** - Easy dependency management

Spring Boot Architecture



Your First Spring Boot Application

```
@SpringBootApplication
public class MyApplication {

  public static void main(String[] args) {
    SpringApplication.run(MyApplication.class, args);
  }
}
```

What happens?

1. Creates IoC container
2. Scans for components (@Controller, @Service, @Repository)
3. Configures beans automatically
4. Starts embedded server on port 8080

Spring Boot Starters

Starters are pre-configured dependency bundles.

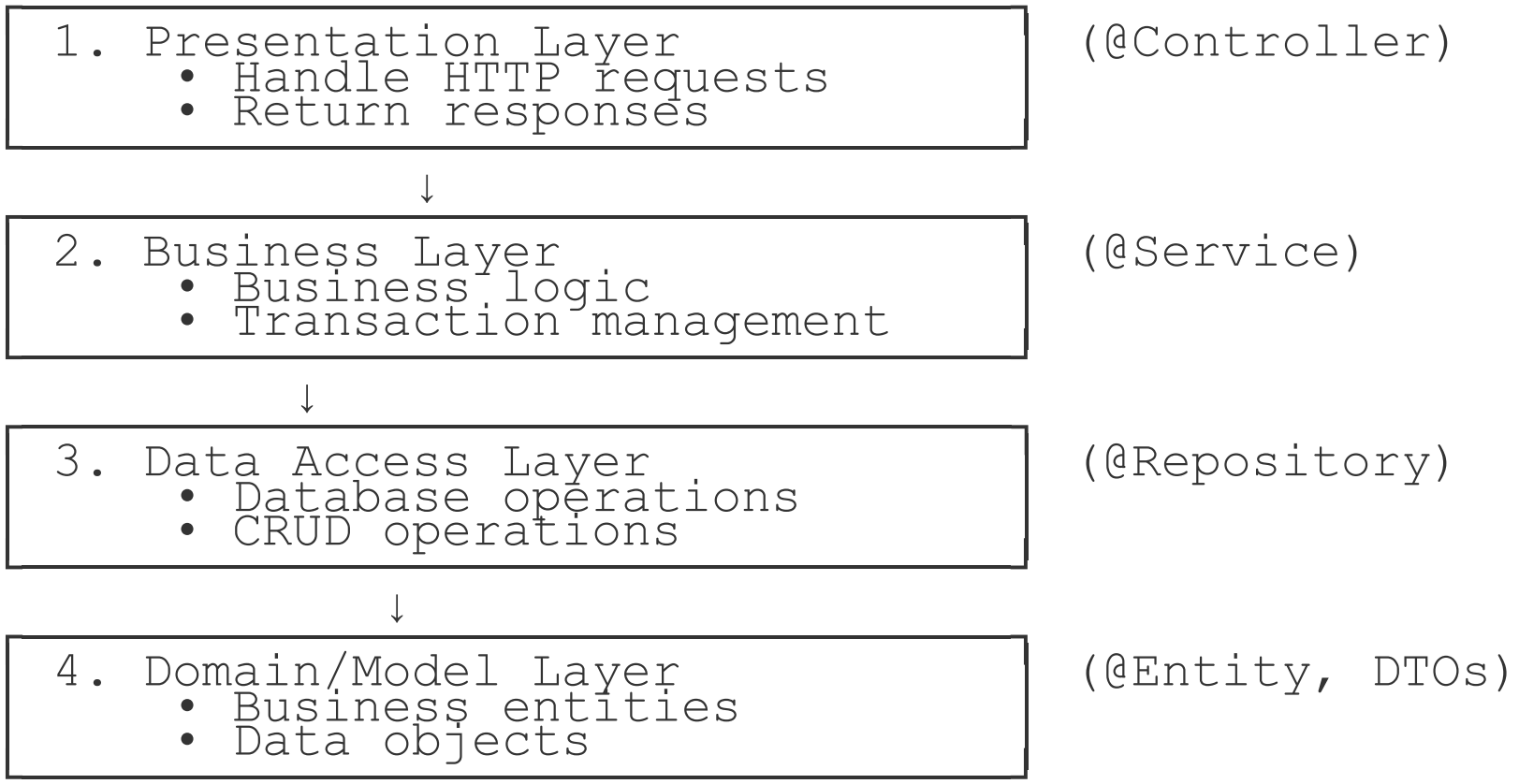
Common Starters:

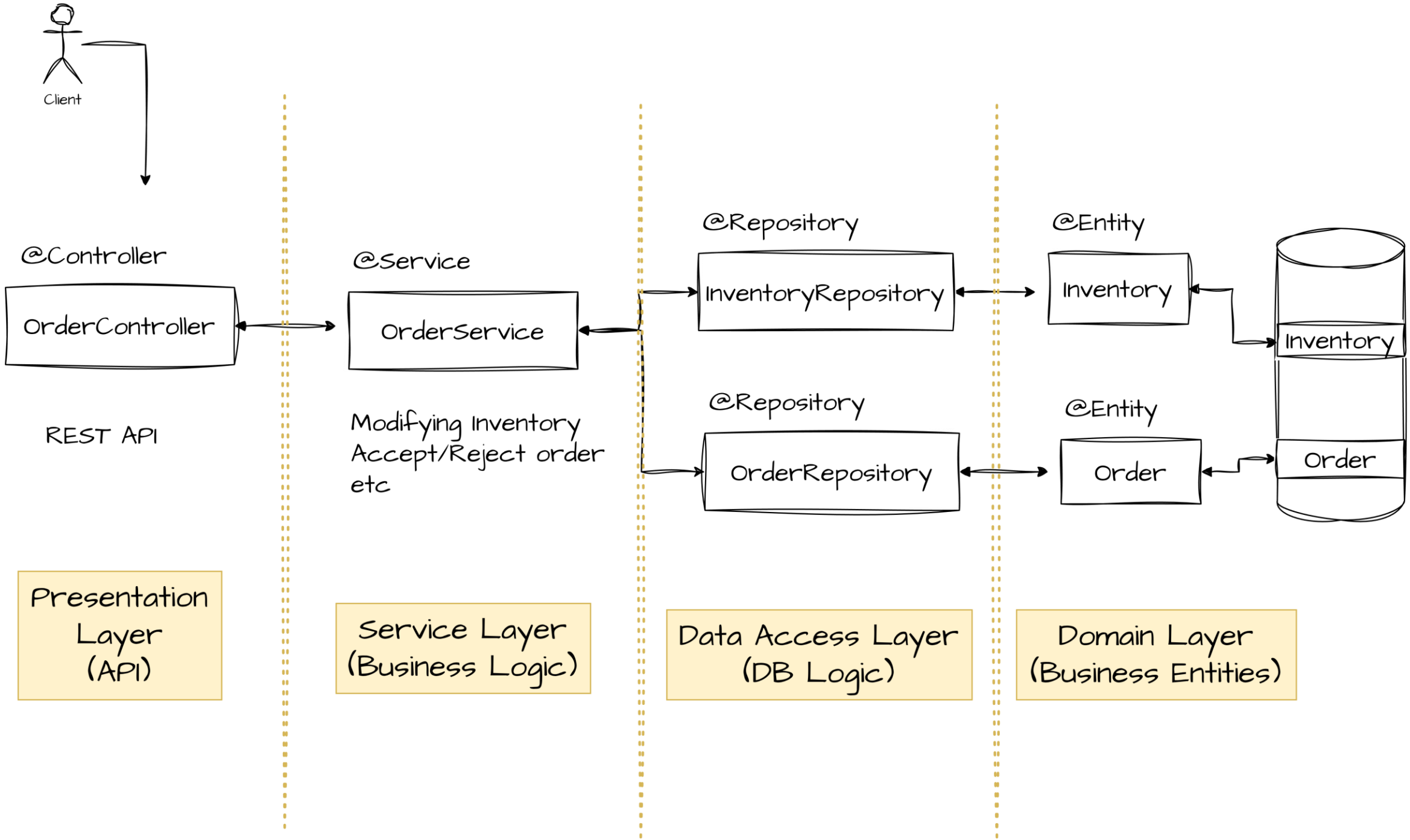
```
<!-- Web Applications -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!-- Database (JPA) -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

Etc.
```

Layered Architecture

Organizes code into distinct layers, each with specific responsibilities.

The (main) 4 Layers:




Client
@Controller
OrderController
REST API
@Service
OrderService
Modifying Inventory
Accept/Reject order
etc
@Repository
InventoryRepository
@Repository
OrderRepository
@Entity
Inventory
@Entity
Order
Inventory
Order
Presentation Layer (API)
Service Layer (Business Logic)
Data Access Layer (DB Logic)
Domain Layer (Business Entities)

Inversion of Control – IoC container

Layman Visualization: The Restaurant Waiter

The Problem (Without IoC):

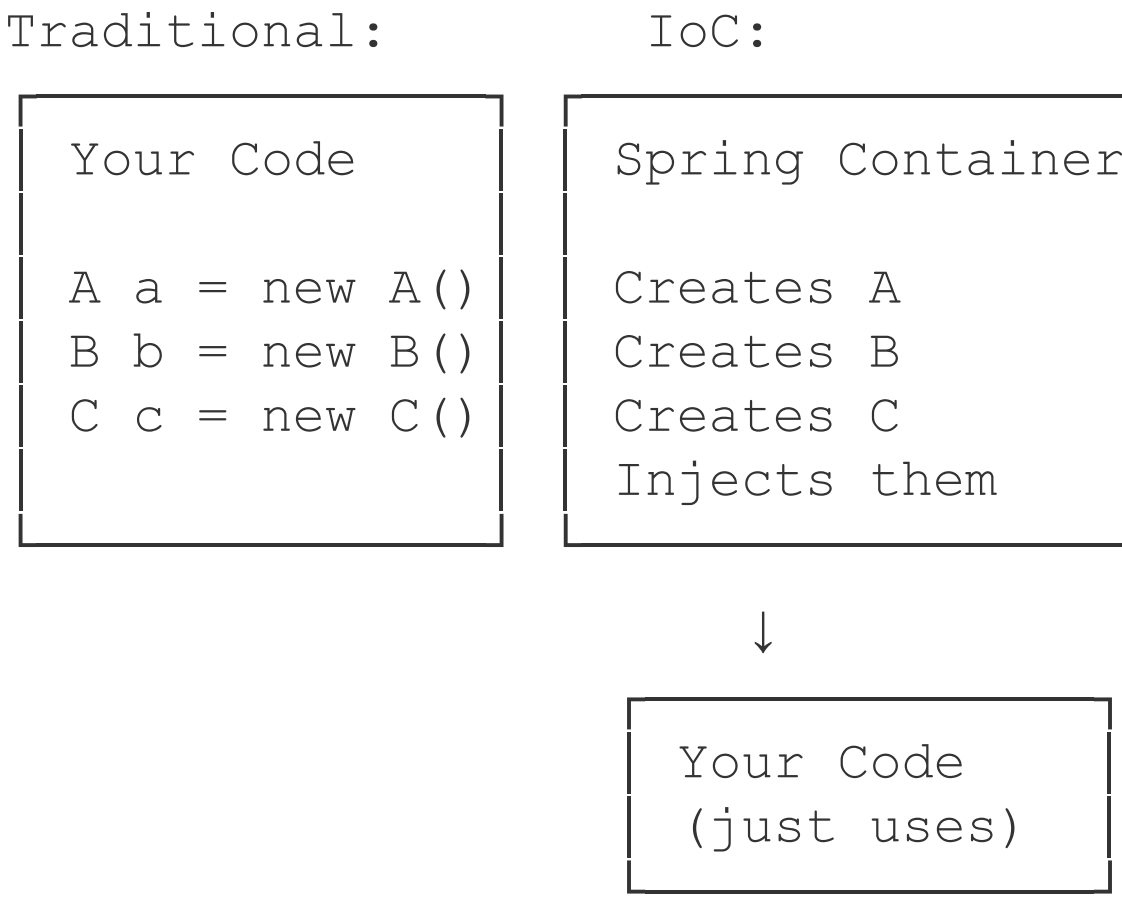
- You walk into a kitchen,
- Cook your own food (create object),
- Find your own ingredients (set dependencies),
- Wash the dishes (manage lifecycle).

The Solution (With IoC):

- You sit at a table (Your Code),
- Tell the waiter what you want (Configuration/Annotations),
- The waiter (Spring Container) brings you a fully prepared dish (Initialized Bean) with all ingredients already added.

What is IoC (Inversion of Control)?

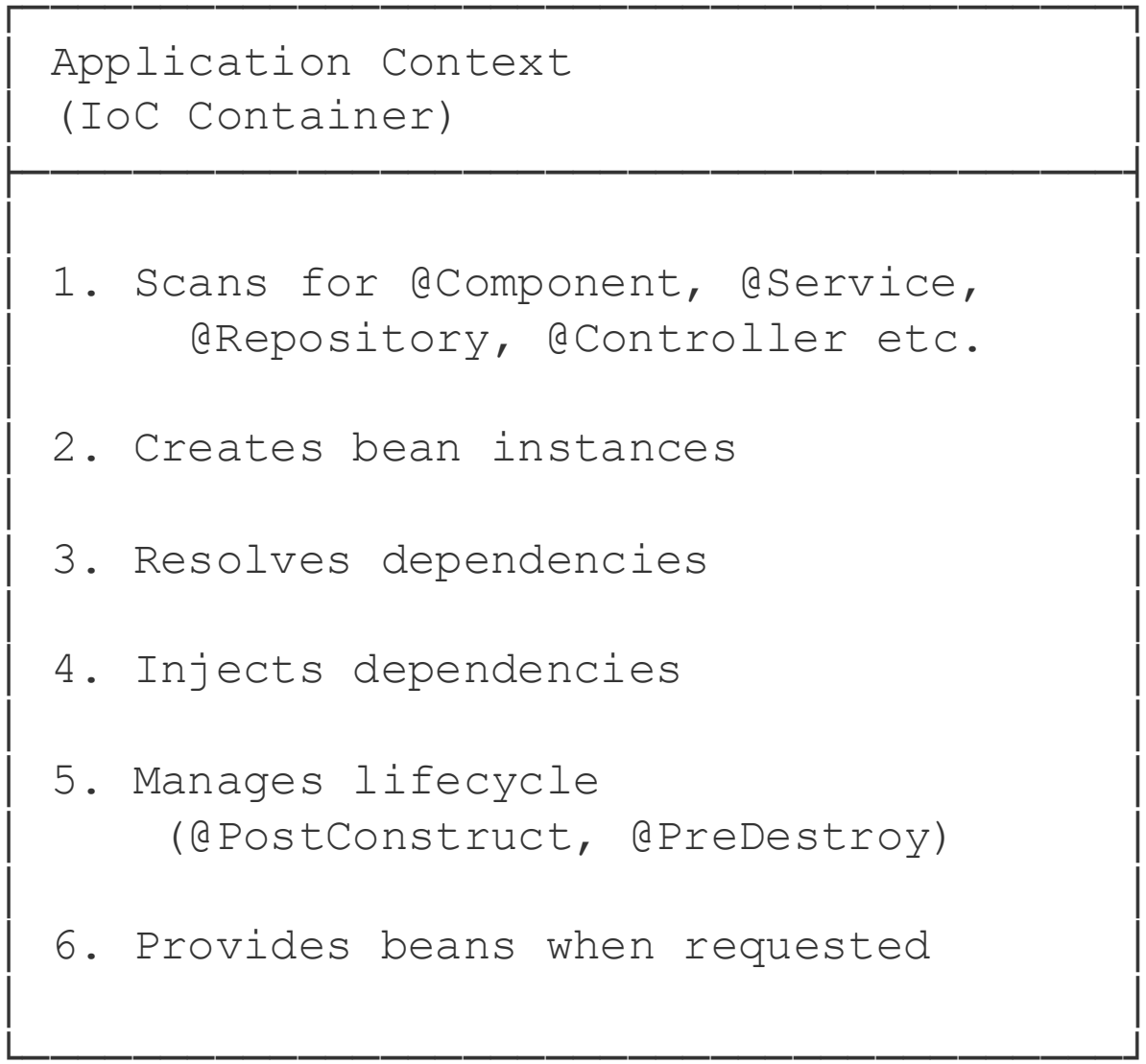
Traditional Approach: Your code controls object creation
IoC Approach: Framework controls object creation



Benefit: "Don't call us, we'll call you"

IoC Container in Spring Boot

The **Application Context** is the IoC container



Dependency Injection (DI)

What is Dependency Injection?

Dependency: Something a class needs to function
Injection: Providing it from outside

```
// Without DI - Class creates its own dependency
public class DeliveryAgent {

  private Vehicle v = new Bike();

  // Tightly coupled (only Bike)
}
```

```
// With DI - Dependency is injected
public class DeliveryAgent {
   private final Vehicle v;

   public DeliveryAgent(Vehicle v) {
      this.v = v;
   }

   // Loosely coupled, testable
   // Use Given Vehicle
}
```

Types of Dependency Injections (DI)

1. Constructor Injection (✅ RECOMMENDED)

```
@Service
public class UserService {
    private final UserRepository repository;

    public UserService(UserRepository repository) {
      this.repository = repository;
    }

}
```

Benefits:

✅ Dependencies are final (immutable)
✅ Easy to test
✅ Clear what dependencies are required

2. Setter Injection

```
@Service
public class NotificationService {

    private SmsService smsService;

    @Autowired(required = false)
    public void setSmsService(SmsService smsService) {
        this.smsService = smsService;
    }

    ...
}
```

Use for: Optional dependencies

3. Field Injection (❌ NOT RECOMMENDED)

```
@Service
public class ProductService {

    @Autowired
    private ProductRepository repository;

    ...
}
```

Problems:

❌ Can't make fields final
❌ Hard to test
❌ Hides dependencies

****Recommendation:**** Always use constructor injection

Spring Beans & Scopes

What are Beans?

Bean: An object managed by Spring's IoC container

```
// Regular Java Object (NOT a bean)
public class Calculator {
    public int add(int a, int b) { return a + b; }
}

Calculator calc = new Calculator(); // You create it
```

```
// Spring Bean (Managed by Spring)
@Component
public class Calculator {
    public int add(int a, int b) { return a + b; }
}
// Spring creates and manages it
```

Creating Beans

Method 1: Stereotype Annotations

```
@Component // Generic component
public class EmailValidator { }

@Service // Business logic
public class UserService { }

@Repository // Data access
public class UserRepository { }

@Controller // Web controller
public class UserController { }
```

Method 2: @Bean in @Configuration

```
@Configuration
public class AppConfig {

    @Bean
    public Calculator calculator() {
      return new Calculator();
    }

}
```

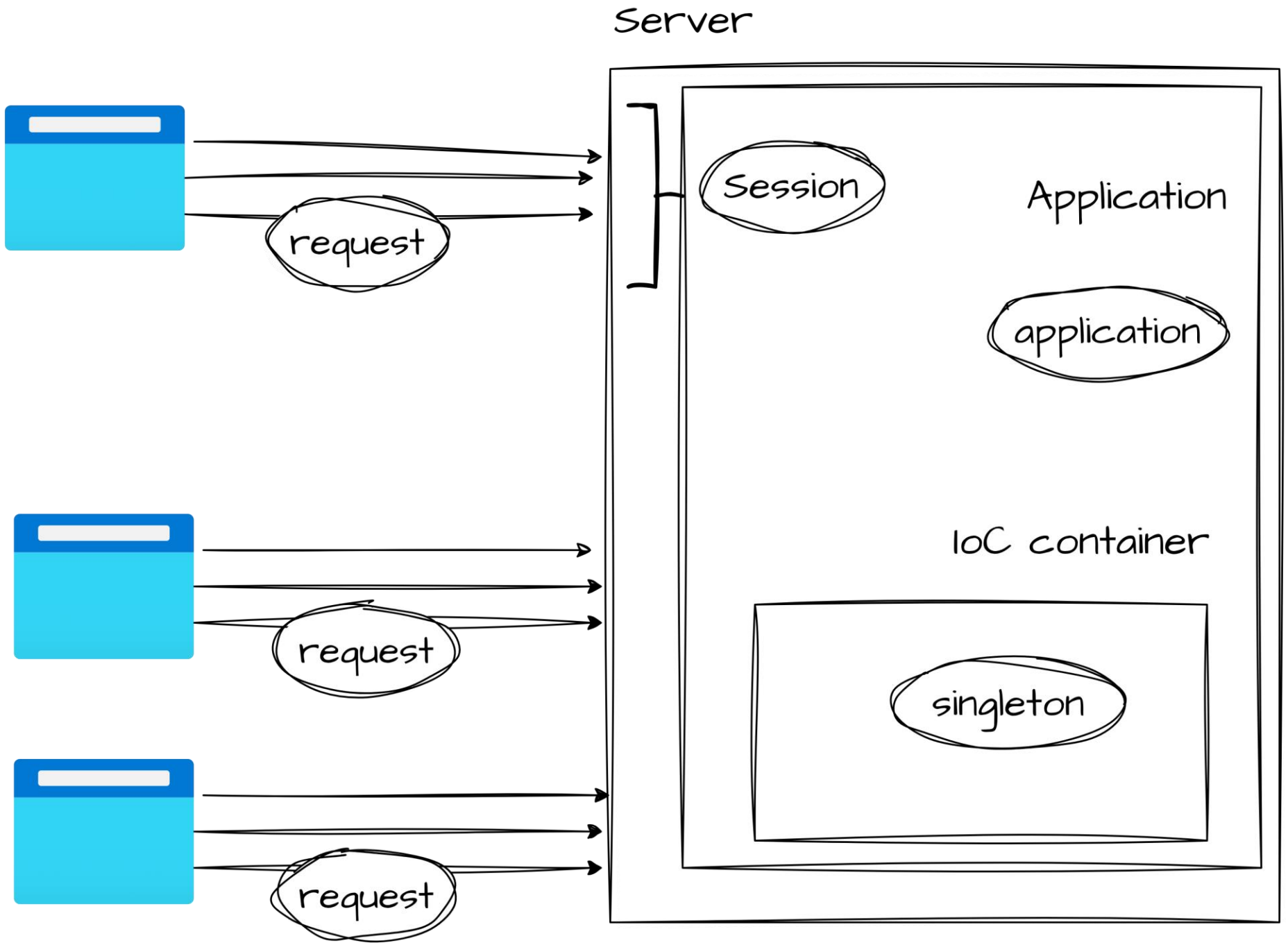
Bean Scopes

Scope: Defines how many instances Spring creates and how long they live

Available Scopes:

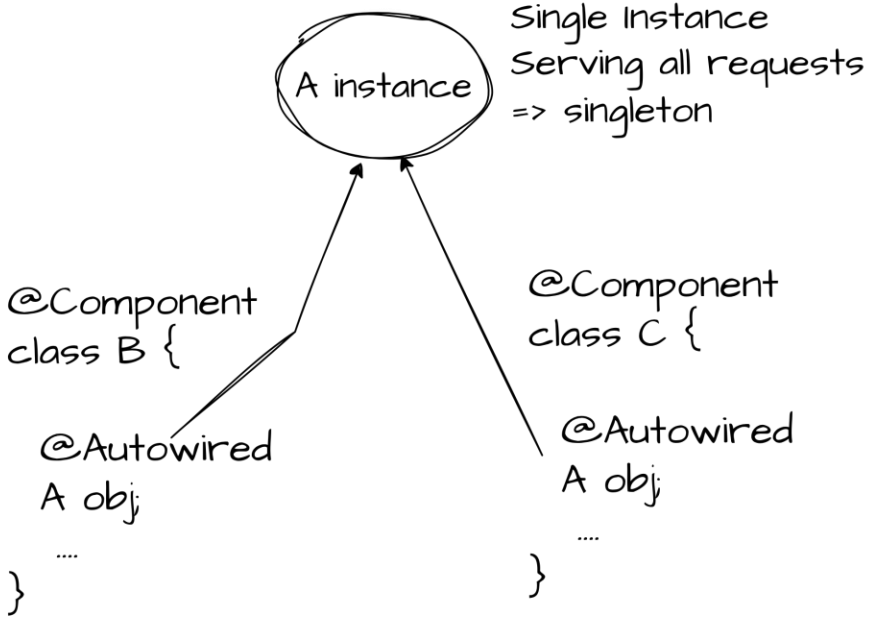
Scope	Instances	Lifecycle	Use Case
Singleton	ONE per container	Application lifetime	Services, Utilities
Prototype	NEW each request	Not managed by Spring	Stateful objects
Request	ONE per HTTP request	Request duration	Request data
Session	ONE per HTTP session	Session duration	User preferences
Application	ONE per ServletContext	Application lifetime	Shared app data

Singleton : Is the default scope for beans.


Server
Session
Application
application
IoC container
singleton
request
request
request

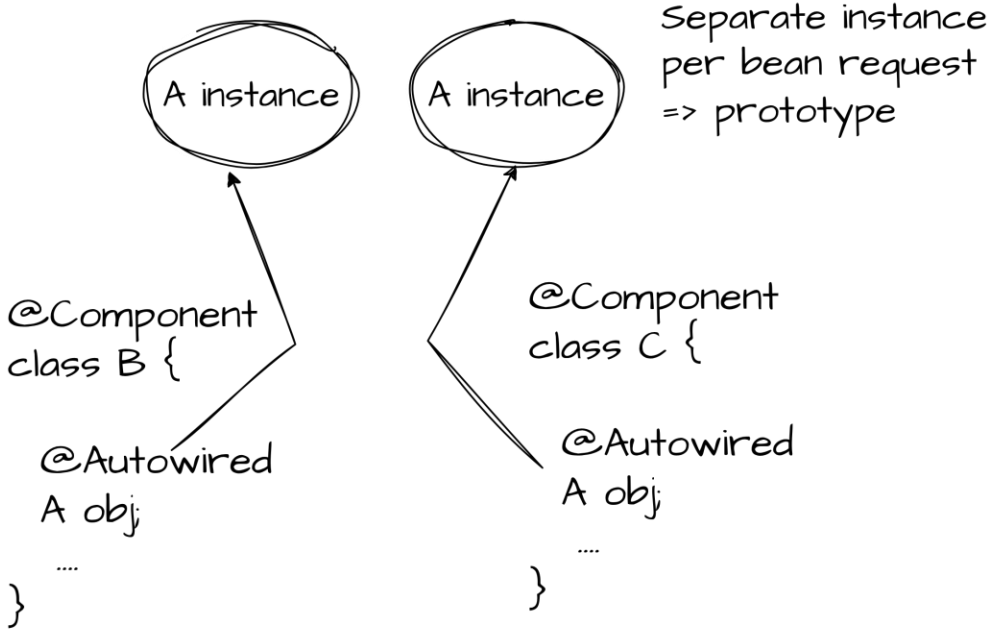
Singleton

```
@Component
@Scope("singleton")
class A {
   ....
}
```



Prototype

```
@Component
@Scope("prototype")
class A {
   ....
}
```

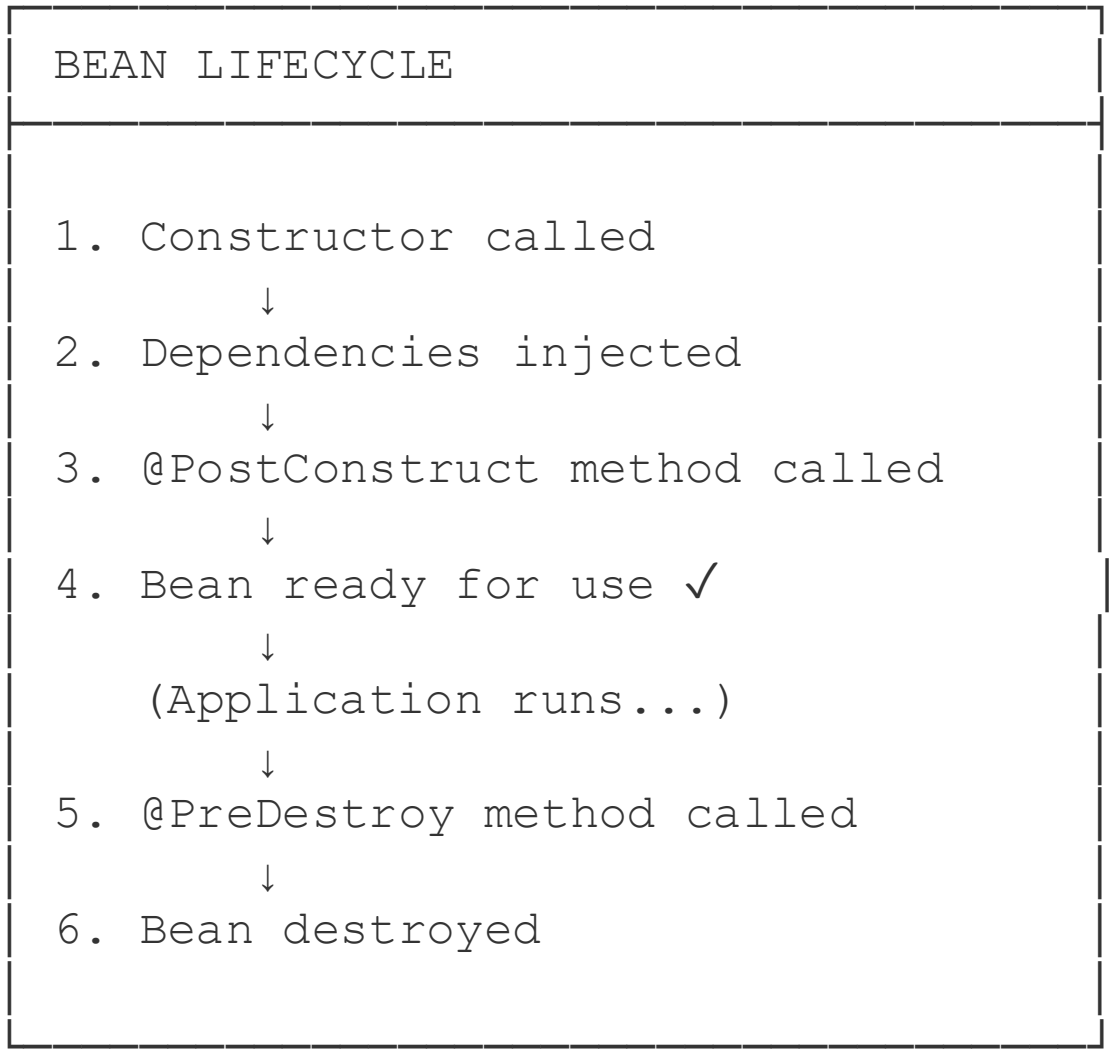


Example : Prototype Scope

NEW instance every time bean is requested

```
@Component
@Scope("prototype")
public class ShoppingCart {
    private List<String> items = new ArrayList<>();

    public void addItem(String item) {
        items.add(item);
    }
    ...
}
```

Bean Lifecycle

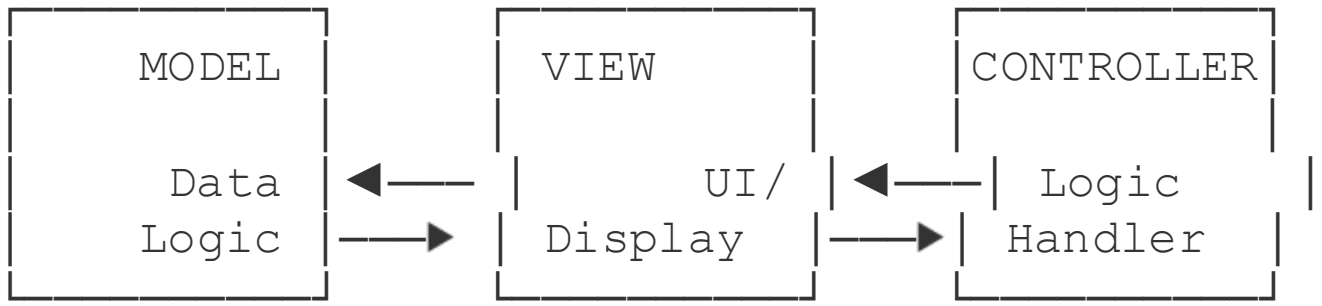


Model View Controller (MVC)

What is MVC?

MVC = Model-View-Controller

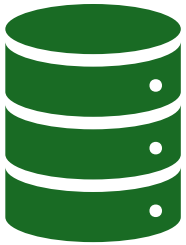
A design pattern that separates application into 3 components:



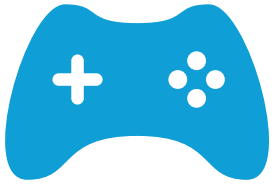
Restaurant Analogy:

- **Controller** = Waiter (takes order, brings food)
- **Model** = Kitchen (prepares food)
- **View** = Plate (how food is presented)

View - UI and Data Display

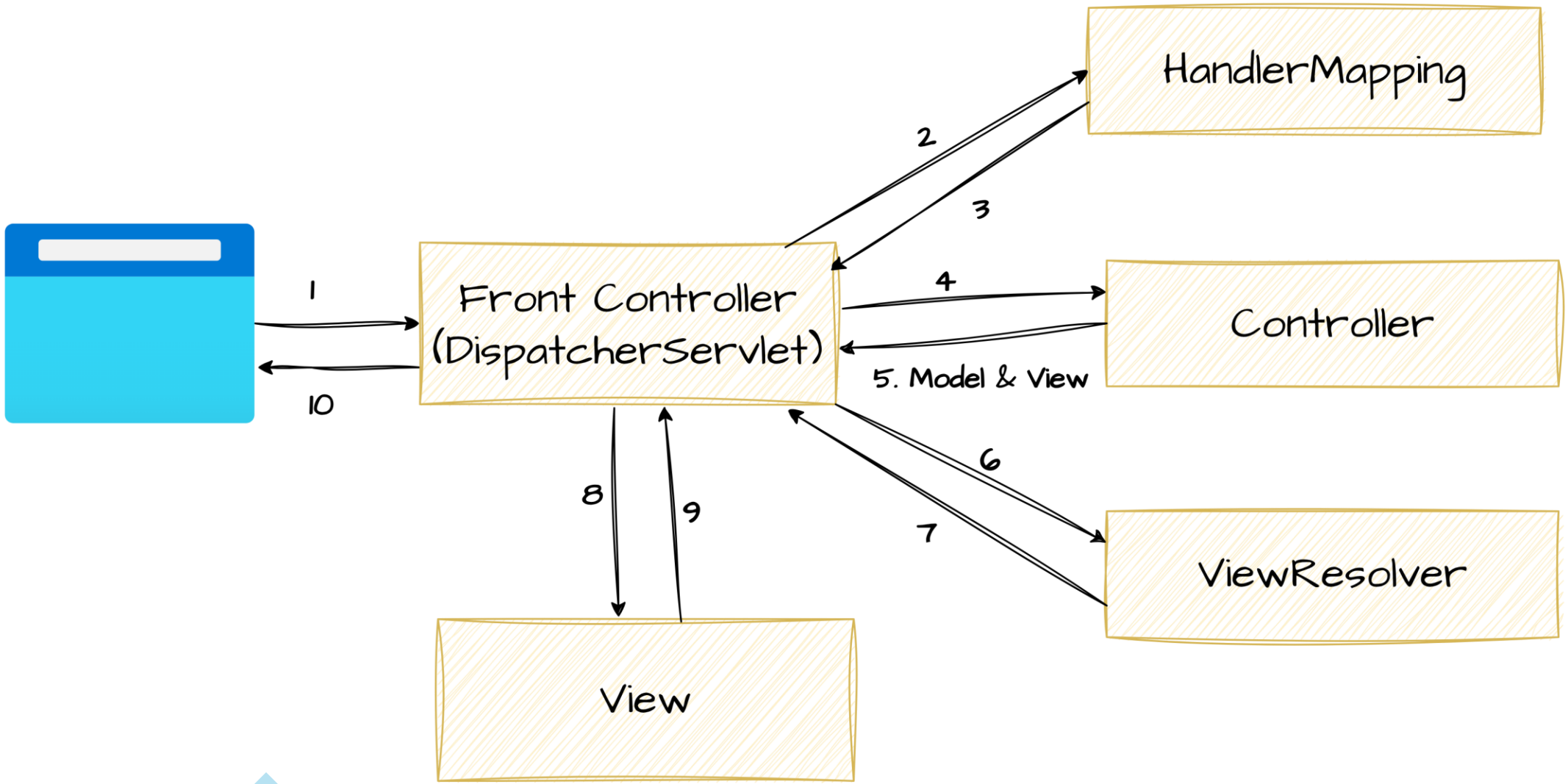


Model – Data and Business Logic



Controller – Intermediary, processes user inputs and updates the model and view.

Spring MVC Flow



Spring MVC Flow

① Browser → HTTP Request

↓

② DispatcherServlet (Front Controller)

↓

③ Handler Mapping (Find controller)

↓

④ CONTROLLER (@Controller)

- Calls Service (Model)
- Returns view name + data

↓

⑤ View Resolver (Find JSP file)

↓

⑥ VIEW (JSP) renders HTML

↓

⑦ HTML Response → Browser

Summary

DispatcherServlet intercepts incoming requests.

Retrieves the handler mapping and forwards the request to the controller.

The controller processes the request and returns a ModelAndView object.

DispatcherServlet uses the view resolver to render the appropriate view.

MVC - Components

1. Model: Represents data and business logic

```
// Entity
@Entity
public class Student {
    @Id
    private Long id;
    private String name;
    private String email;
    private int age;
    // Getters and setters
}

// Service (Business Logic)
@Service
public class StudentService {
    public List<Student> getAllStudents() {
        // Business logic
    }
}
```

2. View (JSP): Displays data to user

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
 <head>
    <title>Students</title>
 </head>
 <body>
   <h1>Student List</h1>
   <table>
     <tr> <th>ID</th> <th>Name</th> <th>Email</th> </tr>
     <c:forEach items="${students}" var="student">
        <tr>
          <td>${student.id}</td>
          <td>${student.name}</td>
          <td>${student.email}</td>
        </tr>
     </c:forEach>
   </table>
 </body>
</html>
```

3. Controller : Handles HTTP requests

```java
@Controller
@RequestMapping("/students")
public class StudentController {

    private final StudentService studentService;

    public StudentController(StudentService studentService) {
            this.studentService = studentService;
    }

    @GetMapping
    public String listStudents(Model model) {
        // Get data from Model
        List<Student> students = studentService.getAllStudents();

        // Add to view model
        model.addAttribute("students", students);

        // Return view name
        return "students"; // → students.jsp
    }
}
```

JSP - Rendering

1

request for /

```java
@Controller
@RequestMapping("/")
public class HomeController {

    @GetMapping("/")
    public String home(Model model) {
        model.addAttribute("pageTitle", "Home");
        model.addAttribute("welcomeMessage",
                "Welcome to the MVC World !!!");

        return "home";  // Resolves to /WEB-INF/views/home.jsp
    }
}
```

2

model

pageTitle	Home
welcomeMessage	Welcome to the MVC World !!!

```jsp
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>${pageTitle}</title>
</head>
<body>
    <div class="container">
        <h1>${welcomeMessage}</h1>
    </div>
</body>
</html>
```

home.jsp

3

4

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Home</title>
</head>
<body>
    <div class="container">
        <h1>Welcome to the MVC World !!!</h1>
    </div>
</body>
</html>
```

After Rendering the Java Server Page (JSP)