Module 2: Application Architecture and Frameworks

Introduction to Spring application Architecture

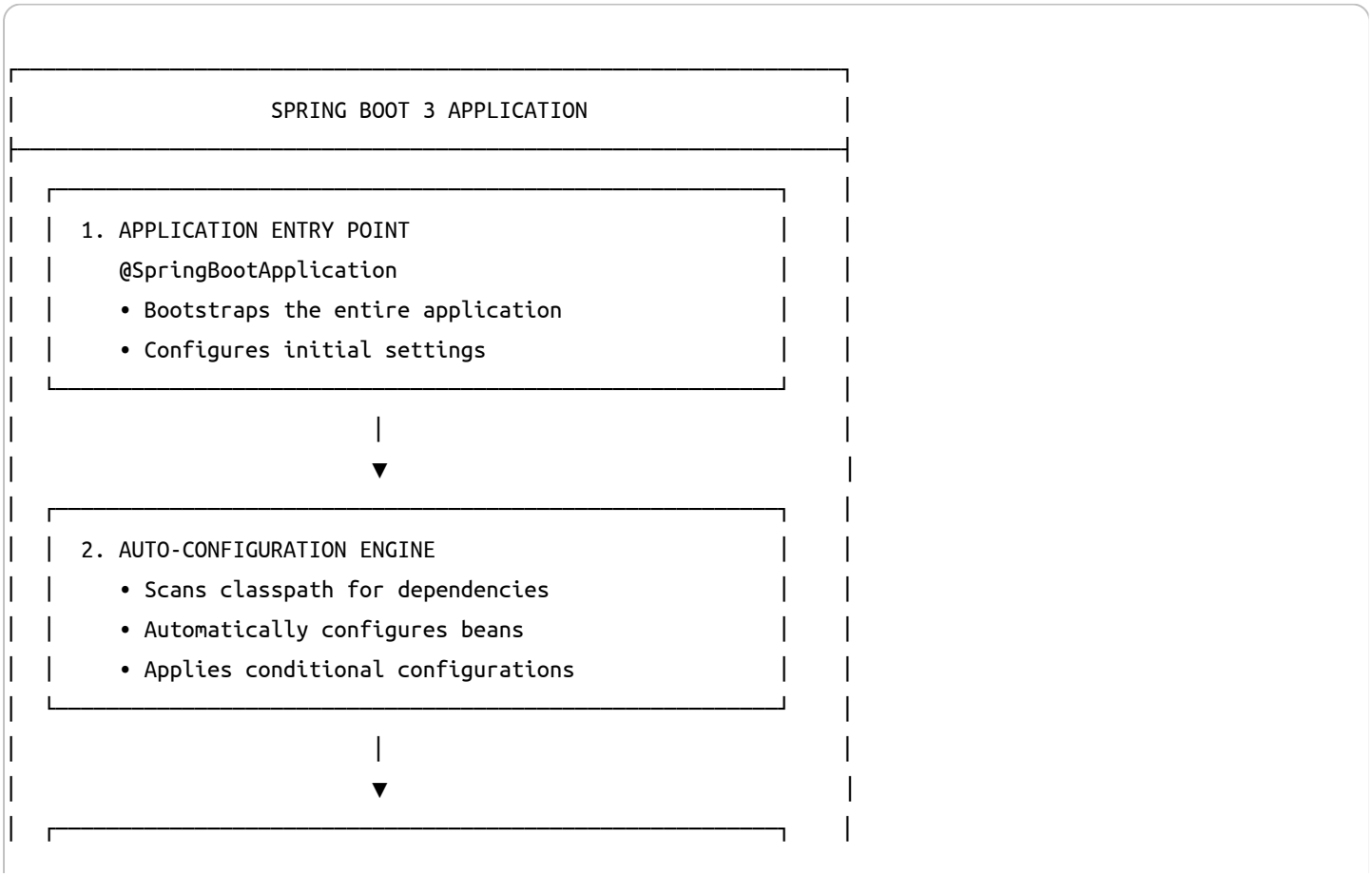
Spring Boot 3 is a powerful framework that simplifies the development of Java applications by providing a well-structured architecture. It builds upon the Spring Framework 6 and introduces a modular, layered approach that makes application development faster and more efficient.

Key Features of Spring Boot Architecture:

- Convention over Configuration
- Auto-configuration based on dependencies
- Embedded web servers (no external deployment needed)
- Production-ready features out of the box
- Dependency Injection for loose coupling

Overall Architecture Overview

Spring Boot follows a **layered architecture pattern** where different components have specific responsibilities and interact in a well-defined manner.



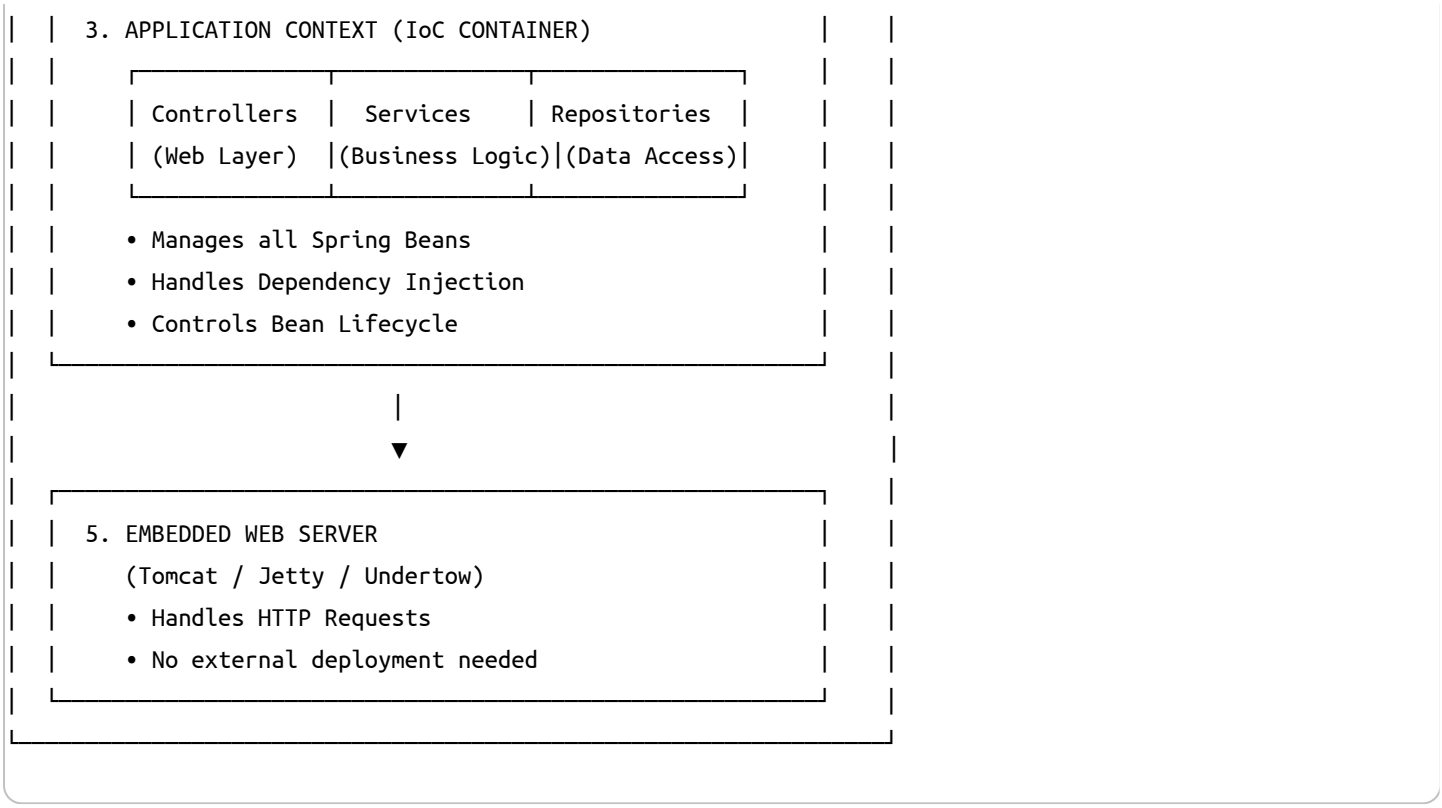


Diagram Explanation:

- The **Application Entry Point** is the main class annotated with `@SpringBootApplication`, which bootstraps the application.
- The **Auto-Configuration Engine** automatically configures beans based on the dependencies present in the classpath.
- The **Application Context (IoC Container)** manages all the Spring beans, handling dependency injection and bean lifecycle.
- The **Embedded Web Server** (like Tomcat, Jetty, or Undertow) handles incoming HTTP requests without needing an external server.

Inversion of Control (IoC)

Inversion of Control (IoC) is one of the most fundamental concepts in Spring Boot. It's a design principle that makes your code more flexible, testable, and maintainable by "inverting" the traditional flow of control in your application.

Key Points:

- IoC is a **design principle**, not a technology
- Spring Boot implements IoC through its **IoC Container**
- **Dependency Injection (DI)** is the primary mechanism to achieve IoC
- It makes your code **loosely coupled** and **easier to test**

What is Inversion of Control (IoC)?

Inversion of Control (IoC): Instead of your code creating and managing objects, you give that control to a framework (Spring Boot). The framework creates objects and injects them where needed.

The Problem IoC Solves

Let’s understand the problem with a real-world example with and with out IoC.

Without IoC - Tightly Coupled Code

```
/**
 * BAD EXAMPLE - Without IoC
 * Tight coupling between classes
 */
// Database access class
class MySQLDatabase {
    public void save(String data) {
        System.out.println("Saving to MySQL: " + data);
    }
}

// Service class that uses database
class UserService {

    // Problem 1: UserService creates its own dependency
    private MySQLDatabase database = new MySQLDatabase();

    public void createUser(String username) {
        // Use the database
        database.save(username);
    }
}

// Main application
public class Application {
    public static void main(String[] args) {
        UserService userService = new UserService();
        userService.createUser("John");
    }
}
```

Problems with This Approach:

- TIGHT COUPLING : UserService is hardwired to MySQLDatabase

- DIFFICULT TO TEST : Can’t replace MySQLDatabase with a test version, Always hits real database during tests
- HARD TO CHANGE : What if you want to switch to PostgreSQL? Must modify UserService code.
- CODE DUPLICATION : Every service creates its own MySQLDatabase instance(object), Wastes memory and resources

With IoC - Loosely Coupled Code

```
/**
 * GOOD EXAMPLE - With IoC
 * Loose coupling through dependency injection
 */

// Database interface (abstraction)
interface Database {
    void save(String data);
}

// MySQL implementation
class MySQLDatabase implements Database {
    public void save(String data) {
        System.out.println("Saving to MySQL: " + data);
    }
}

// PostgreSQL implementation
class PostgreSQLDatabase implements Database {
    public void save(String data) {
        System.out.println("Saving to PostgreSQL: " + data);
    }
}

// Service class - now flexible!
class UserService {

    // UserService doesn't create MySQLDatabase object anymore
    // It receives it from outside (IoC Container) based on configuration
```

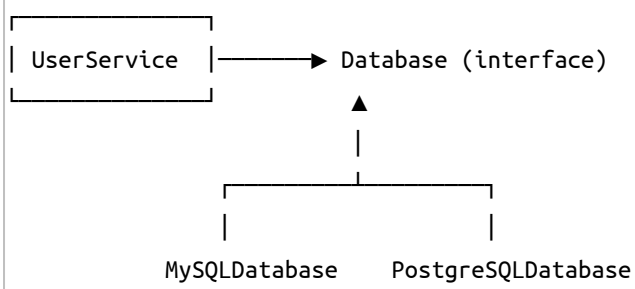
```
    private Database database;

    // Constructor injection - Spring IoC container provides the database
    public UserService(Database database) {
        this.database = database;
    }

    public void createUser(String username) {
        database.save(username);
    }
}
```

Benefits of IoC Approach:

- LOOSE COUPLING : UserService works with any Database



- EASY TO TEST : Can inject mock database for testing
- FLEXIBLE : Switch databases without changing UserService
- REUSABLE : One database instance can be shared by all services, Better resource management.

Dependency Injection (DI)

Dependency Injection (DI) is the primary technique used to implement IoC. It's how the container provides dependencies to your beans.

What is Dependency Injection?

```
class A {
   B b = new B();  // A creates B

   // A depends on B
 }
```

Instead of directly creating B inside A, make it receive B from outside.

```
class A {
    B b;

    A(B b) {  // Someone else gives B to A
      this.b = b;
    }
}
```

- **Dependency** = Something a class needs (B)
- **Injection** = Providing it from outside

Why Use Dependency Injection?

- **Decouples Components**: Classes don’t need to know how to create their dependencies.
- **Easier Testing**: You can inject mock dependencies during testing.
- **Improved Maintainability**: Changing a dependency doesn’t require changes in dependent classes.

Types of Dependency Injection

Spring Boot supports three types of dependency injection. Let’s examine each with examples.

1. Constructor Injection
2. Setter Injection
3. Field Injection

1. Constructor Injection (RECOMMENDED)

All dependencies are provided through the class constructor and it is the preferred method.

```
/**
 * CONSTRUCTOR INJECTION
 * Dependencies injected through constructor
 * BEST PRACTICE - Use this approach!
 */

@Service
public class UserService {

    private final Database database;
    private final EmailService emailService;

    // Constructor with dependencies as parameters
    // Spring automatically calls this and provides dependencies,
    // @Autowired is optional here if there is only one constructor.
```

```
    public UserService(Database database, EmailService emailService) {
        this.database = database;
        this.emailService = emailService;
    }

    public void createUser(String username) {
        database.save(username);
        emailService.sendWelcomeEmail(username);
    }
}
```

2. Setter Injection

Dependencies are provided through setter methods after the object is created.

```
/**
 * SETTER INJECTION
 * Dependencies injected through setter methods
 */

@Service
public class UserService {

    private final Database database;
    private final EmailService emailService;

    // Setter for Database
    @Autowired
    public void setDatabase(Database database) {
        this.database = database;
    }

    // Setter for EmailService
    @Autowired
    public void setEmailService(EmailService emailService) {
        this.emailService = emailService;
    }

    public void createUser(String username) {
        database.save(username);
        emailService.sendWelcomeEmail(username);
    }
}
```

3. Field Injection

Fields are injected directly. This approach is generally discouraged due to several drawbacks such as difficulty in testing and lack of immutability.

```
/**
 * FIELD INJECTION
 * Dependencies injected through fields directly
 * NOT RECOMMENDED - Avoid this approach
 */

@Service
public class UserService {

    @Autowired
    Database database;

    @Autowired
    EmailService emailService;

    public void createUser(String username) {
        database.save(username);
        emailService.sendWelcomeEmail(username);
    }
}
```

Bean scopes and configuration

What are Beans?

Bean: A bean is simply an object that is created and managed by Spring Boot's IoC container. Think of beans as **"special objects"** that Spring takes care of for you. In Spring Boot, a bean is any Java object that is instantiated, assembled, and managed by the Spring IoC container. Beans are the backbone of a Spring application, and they represent the various components that make up your application, such as services, repositories, controllers, and more.

```
/**
 * Example of a Spring Bean
 */
@Component
public class MyService {
    public void performAction() {
        System.out.println("Action performed!");
    }
```

```
}
```

When you define a class and annotate it with specific Spring annotations (like `@Component`, `@Service`, `@Repository`, or `@Controller`), Spring automatically detects these classes during **component scanning** and creates instances of them as beans in the application context.

Bean Scopes

In Spring Boot, beans can have different scopes that define their lifecycle and visibility within the application context. The most common bean scopes are:

- **Singleton (Default)**: A single instance of the bean is created and shared across the entire application context. This is the default scope in Spring.
- **Prototype**: A new instance of the bean is created each time it is requested from the container.
- **Request**: A new instance of the bean is created for each HTTP request. This scope is only valid in web applications.
- **Session**: A new instance of the bean is created for each HTTP session. This scope is also only valid in web applications.
- **Application**: A single instance of the bean is created for the lifecycle of a web application.

Configuring Beans

You can configure beans in Spring Boot using Java-based configuration with the `@Configuration` and `@Bean` annotations.

```
/**
 * Example of Bean Configuration
 */
@Configuration
public class AppConfig {

    @Bean
    @Scope("singleton") // Default scope
    public MyService myService() {
        return new MyService();
    }

}
```

In this example, the `AppConfig` class is marked with `@Configuration`, indicating that it contains bean definitions. The `myService` method is annotated with `@Bean`, which tells Spring to create and manage an instance of `MyService` as a bean in the application context. `EmailService` and `Database` can also be defined similarly. And finally, these beans can be injected into other components using dependency injection.

Combined Example of Dependency Injection and Bean Declaration

Here is the full length working example of Dependency Injection and Bean declaration in Spring Boot:

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

@Service
class UserService {

    private final Database database;
    private final EmailService emailService;

    @Autowired // Constructor Injection, @Autowired is optional here it is the only constructor.
    public UserService(Database database, EmailService emailService) {
        this.database = database;
        this.emailService = emailService;
    }

    public void createUser(String username) {
        database.save(username);
        emailService.sendWelcomeEmail(username);
    }
}

interface Database {
    void save(String data);
}

@Service
class MySQLDatabase implements Database {
    public void save(String data) {
        System.out.println("Saving to MySQL: " + data);
    }
}

@Service
class EmailService {
    public void sendWelcomeEmail(String username) {
        System.out.println("Sending welcome email to: " + username);
    }
```

```
}

@Configuration
class AppConfig {

    @Bean
    public Database database() {
        return new MySQLDatabase();
    }
}

@SpringBootTest
class DependencyInjectionExampleTests {

    @Autowired
    private UserService userService;

    @Test
    void testCreateUser() {
        userService.createUser("TestUser");
    }
}
```

Modular architecture and clean code practices

Modular Architecture

Modular architecture is a design approach that divides an application into separate, independent modules, each responsible for a specific functionality. This approach enhances maintainability, scalability, and reusability of code.

Following are some key principles of modular architecture:

- **Separation of Concerns or Single Responsibility**: Each module should have a single responsibility and should not mix different functionalities.
- **Encapsulation**: Modules should hide their internal implementation details and expose only necessary interfaces.
- **Loose/Low Coupling**: Modules should interact with each other through well-defined interfaces, minimizing dependencies between them.
- **High Cohesion**: Related functionalities should be grouped together within the same module to enhance clarity and maintainability.

Real-World Analogy

Think of a Restaurant:

```
┌──────────────────────────────────────────┐
│  Kitchen  |  Dining  |  Cashier   |  Delivery  |
│  (Cook)   |  (Serve) |  (Payment) |  (Ship)    |
└──────────────────────────────────────────┘
```

- Each area has ONE job (Single Responsibility)
- Kitchen doesn't handle payments (Separation of Concerns)
- Changing delivery doesn't affect cooking (Low Coupling)
- All cooking happens in kitchen (High Cohesion)

Same applies to your Spring Boot application!

Clean Code Practices

Clean code practices are essential for writing code that is easy to read, understand, and maintain. Here are some key clean code principles:

- **Meaningful Names**: Use descriptive and meaningful names for variables, functions, classes, and modules.
- **Single Responsibility Principle**: Each function or class should have one reason to change, meaning it should only have one responsibility.
- **Consistent Formatting**: Follow a consistent coding style and formatting guidelines to enhance readability.
- **Avoid Magic Numbers**: Use named constants instead of hardcoding numbers in the code.
- **Write Tests**: Implement unit tests and integration tests to ensure code correctness and facilitate refactoring.
- **Refactor Regularly**: Continuously improve the codebase by refactoring to eliminate duplication and improve structure.
- **Document Code**: Use comments and documentation to explain complex logic and provide context where necessary.

Here are few other clean code practices applicable to Spring Boot development: (FOR REFERENCE)

- Use Spring stereotypes (`@Service`, `@Repository`, `@Controller`) appropriately to indicate the role of each class.
- Leverage Spring's dependency injection to manage dependencies and promote loose coupling.
- Organize code into packages based on functionality (e.g., `controller`, `service`, `repository`, `model`).
- Use configuration properties (`application.properties` or `application.yml`) to externalize configuration settings.
- Handle exceptions gracefully using Spring's exception handling mechanisms (e.g., `@ControllerAdvice`).
- Utilize Spring Boot's auto-configuration features to reduce boilerplate code.
- Follow RESTful principles when designing APIs with Spring MVC.

MODULAR ARCHITECTURE & CLEAN CODE SUMMARY (for quick reference)

```
┌─────────────────────────────────────────────────────────┐
│         MODULAR ARCHITECTURE & CLEAN CODE SUMMARY       │
├─────────────────────────────────────────────────────────┤
│  1. MODULAR ARCHITECTURE                                │
│     • Break app into independent modules                │
│     • Each module has ONE responsibility                │
│     • Low coupling, high cohesion                       │
│     • Easy to test, maintain, and scale                 │
│                                                         │
│  2. LAYERED ARCHITECTURE                                │
│     • Controller: HTTP/Protocol handling                │
│     • Service: Business logic                           │
│     • Repository: Database access                       │
│     • Domain: Data models                               │
│                                                         │
│  3. PACKAGE STRUCTURE                                   │
│     • Feature-based > Layer-based                       │
│     • Keep related code together                        │
│     • Separate common code                              │
│                                                         │
│  4. CLEAN CODE PRINCIPLES                               │
│     • Meaningful names                                  │
│     • Single responsibility                             │
│     • Small methods                                     │
│     • DRY (Don't Repeat Yourself)                       │
│     • Dependency injection                              │
│                                                         │
│  5. SOLID PRINCIPLES                                    │
│     • Single Responsibility                             │
│     • Open/Closed                                       │
│     • Liskov Substitution                               │
│     • Interface Segregation                             │
│     • Dependency Inversion                              │
│                                                         │
│  6. AVOID COMMON MISTAKES                               │
│     • No business logic in controllers                  │
│     • No direct repository access from controllers      │
│     • Proper exception handling                         │
│                                                         │
└─────────────────────────────────────────────────────────┘
```

MVC architecture using Spring MVC, Integration with templates (JSP)

MVC Architecture Overview

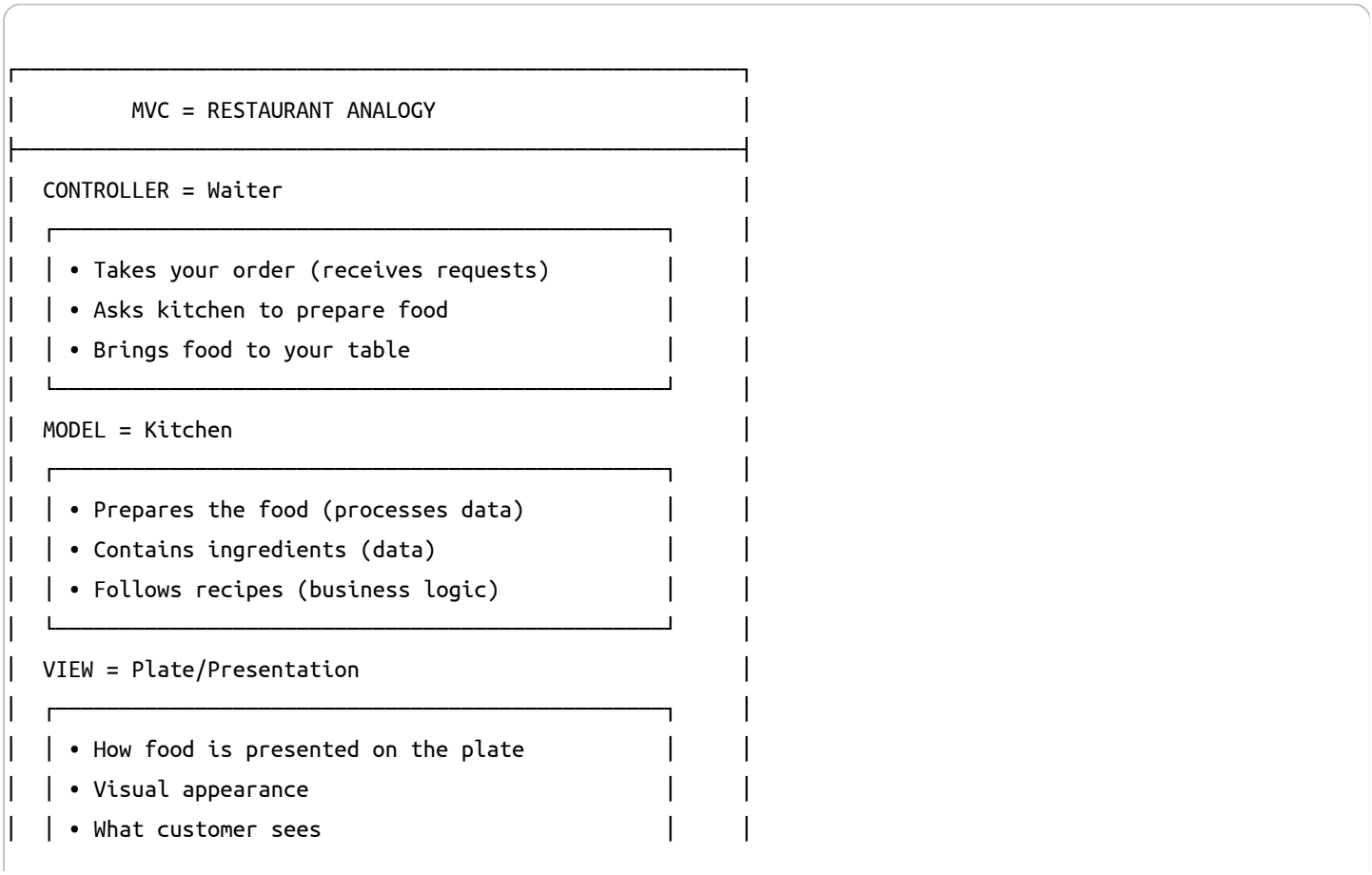
MVC (Model-View-Controller) is a design pattern that separates an application into three main components: Model, View, and Controller. This separation helps manage complexity, promotes organized code, and enhances maintainability.

- Model: Represents the data and business logic of the application. It encapsulates the application's state and behavior.
- View: Responsible for rendering the user interface. It displays data from the model to the user and captures user input.
- Controller: Acts as an intermediary between the Model and View. It processes user input, interacts with the Model, and determines which View to render.

Here is the typical flow in an MVC architecture:

- User interacts with the View (e.g., clicks a button).
- The View sends the user input to the Controller.
- The Controller processes the input, interacts with the Model to retrieve or update data.
- The Model notifies the View of any changes in data.
- The View updates the user interface based on the Model's state.

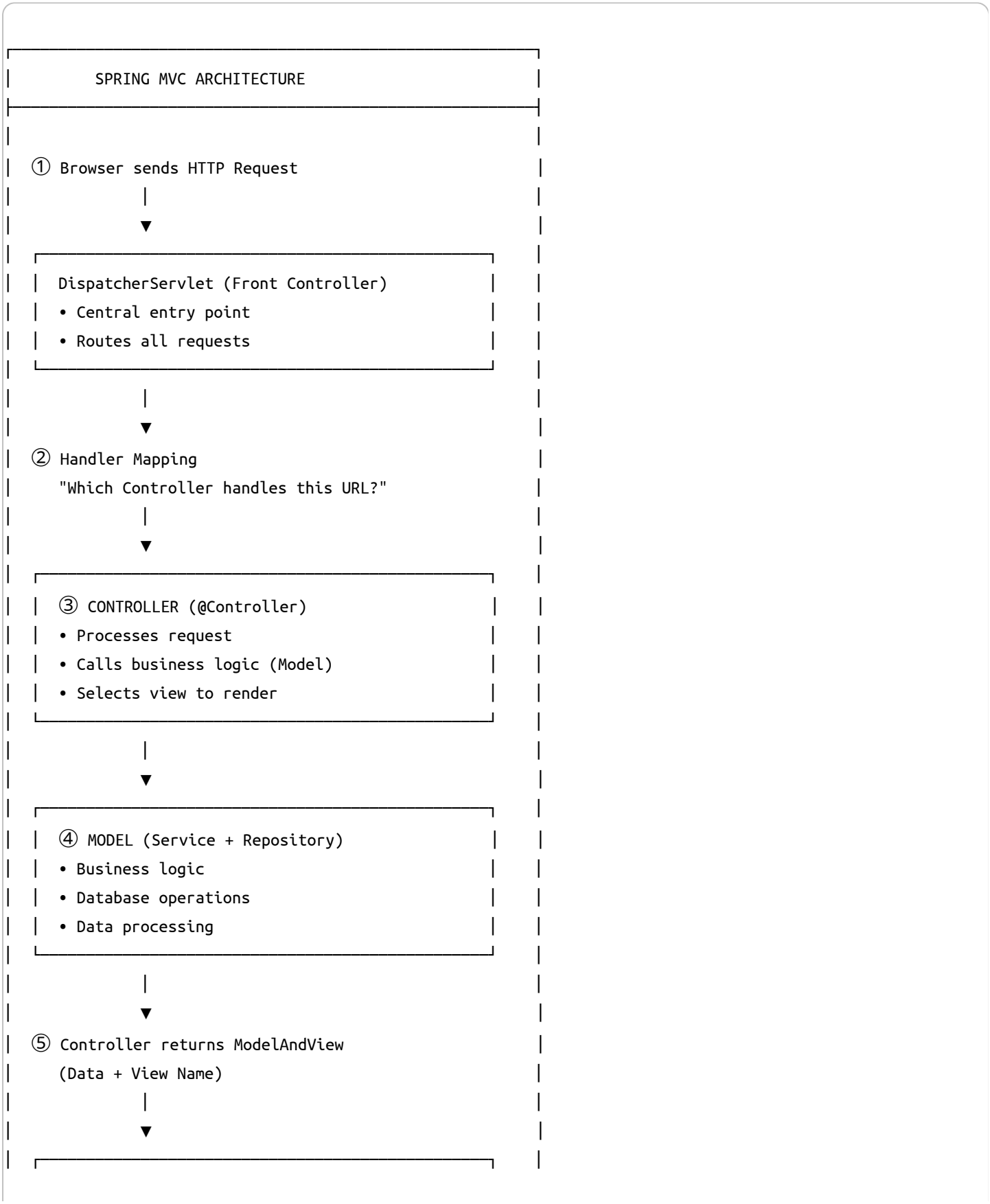
Real-World Analogy

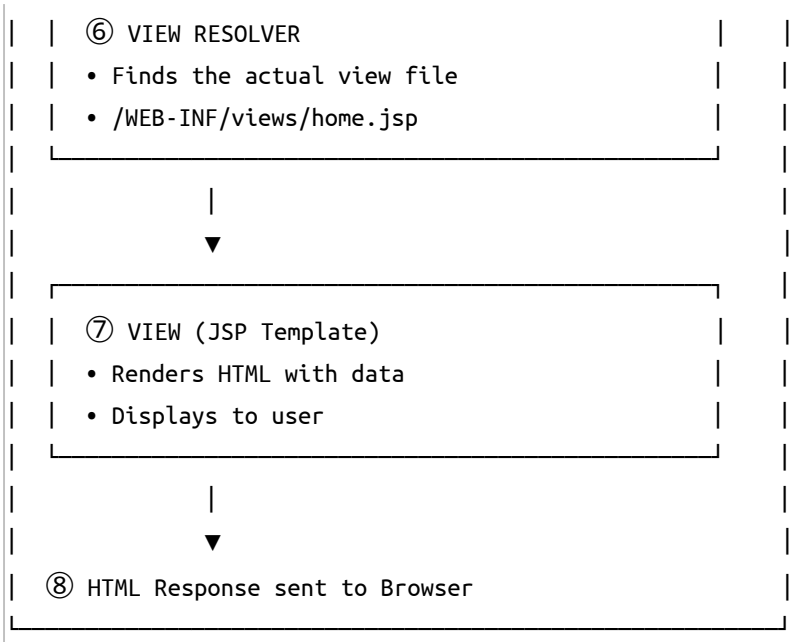


```
│  └──────────────────────────────────────────┘     │
└──────────────────────────────────────────────────┘
```

What is Spring MVC?

Spring MVC is Spring's framework for building web applications using the MVC pattern.





Spring MVC Integration with JSP

Spring MVC is a framework that implements the MVC pattern and provides a robust way to build web applications. It integrates seamlessly with JSP (JavaServer Pages) for rendering views.

Setting Up Spring MVC with JSP

1. **Add Dependencies**: Ensure you have the necessary Spring MVC and JSP dependencies in your `pom.xml` file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>javax.servlet.jsp-api</artifactId>
    <scope>provided</scope>
</dependency>
```

2. **Configure View Resolver**: Set up a view resolver to map view names to JSP files.

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Bean
    public InternalResourceViewResolver viewResolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }
}
```

3. **Create Model**: Define a simple model class.

```
public class User {
    private String name;
    // Getters and Setters
}
```

4. **Create Controller**: Implement a controller to handle requests.

```
@Controller
public class UserController {
    @GetMapping("/user")
    public String getUser(Model model) {
        User user = new User();
        user.setName("John Doe");
        model.addAttribute("user", user);
        return "userView"; // Maps to /WEB-INF/views/userView.jsp
    }
}
```

5. **Create JSP View**: Create a JSP file to render the view.

```
<!-- WEB-INF/views/userView.jsp -->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
    <title>User Info</title>
</head>
<body>
    <h1>User Information</h1>
    <p>Name: ${user.name}</p>
</body>
```

```
</html>
```

6. **Run the Application**: Start your Spring Boot application and navigate to `http://localhost:8080/user` to see the rendered JSP view displaying user information.

This completes the overview of Module 2: Application Architecture and Frameworks in Spring Boot 3, covering key concepts such as application architecture, Inversion of Control (IoC), Dependency Injection (DI), bean scopes and configuration, modular architecture, clean code practices, and MVC architecture using Spring MVC with JSP integration.