Introduction to Maven (Java Build Tool)

Maven is a powerful open-source project management and comprehension tool primarily used for Java projects. It automates the build process, dependency management, and project information reporting.

- **Build Tool:** Maven compiles source code, runs tests, creates JAR/WAR files, and generates documentation.
- **Dependency Management:** One of Maven's most significant features is handling project dependencies. Instead of manually downloading and adding JARs, you declare dependencies in a `pom.xml` file, and Maven automatically downloads them from a central repository (Maven Central) and manages their transitive dependencies.
- **Project Object Model (POM):** At the heart of Maven is the `pom.xml` file, an XML configuration that describes the project, its dependencies, plugins, and build lifecycle.
- **Convention over Configuration:** Maven promotes a standard directory structure and build lifecycle, reducing the need for explicit configuration.
- **Lifecycle:** Maven has a well-defined lifecycle (e.g., `validate`, `compile`, `test`, `package`, `install`, `deploy`). Running `mvn install` executes all phases up to `install`.

Example `pom.xml` snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>my-spring-app</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>6.0.11</version>
        </dependency>
        <!-- Add more dependencies as needed -->
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
```

```
                <version>3.8.1</version>
            </plugin>
        </plugins>
    </build>
</project>
```

Spring Framework Overview

The Spring Framework is a powerful, comprehensive, open-source application framework for the Java platform. It provides a robust and flexible architecture for building enterprise-grade applications.

- **Core Principles:** Lightweight, inversion of Control (IoC), Aspect-Oriented Programming (AOP), and a powerful abstraction for various technologies (data access, web, messaging, etc.).
- **Modular:** Spring is designed in a modular fashion, allowing developers to use only the parts they need.
- **POJO-based Development:** Enables building applications with Plain Old Java Objects (POJOs), avoiding vendor lock-in and complex frameworks.

Need of Spring

Before Spring, Java EE development often involved heavy, intrusive frameworks (like EJB 2.x) that required specific interfaces, inheritance hierarchies, and complex XML configurations, leading to tightly coupled and difficult-to-test code.

Spring addresses these challenges by:

- **Reducing Complexity:** Simplifies enterprise application development.
- **Promoting Loose Coupling:** Through Dependency Injection, components are independent.
- **Enhancing Testability:** IoC makes it easy to mock dependencies for unit testing.
- **Providing Abstractions:** Offers consistent APIs over various technologies (JDBC, ORM, JMS, etc.).
- **Increasing Productivity:** Reduces boilerplate code and speeds up development.

Overview of Spring 5 Modules

Spring 5 (and later versions like Spring 6) is structured into several modules, each providing specific functionalities:

- **Spring Core Container:** The fundamental module, providing IoC and Dependency Injection.
- **Spring AOP:** Enables Aspect-Oriented Programming for cross-cutting concerns (e.g., logging, transaction management).
- **Spring Data Access/Integration:**
    - **JDBC:** Simplifies JDBC access.
    - **ORM:** Integration with ORM frameworks like Hibernate, JPA.
    - **OXM:** Object/XML mapping.
    - **JMS:** Java Messaging Service integration.
    - **Transactions:** Declarative transaction management.
- **Spring Web:**
    - **Web MVC:** Building web applications (Model-View-Controller).
    - **WebFlux:** Reactive web stack for asynchronous, non-blocking applications.

- **Spring Security:** Comprehensive security services.
- **Spring Test:** Support for unit and integration testing.
- **Spring Boot:** (Discussed later) Provides rapid application development capabilities on top of Spring.

Inversion of Control (IoC)

IoC is a design principle where the flow of control of a program is inverted compared to traditional imperative programming. Instead of the developer controlling object creation and lifecycle, a framework (the IoC container) takes over.

- **"Don't call us, we'll call you" (Hollywood Principle):** The framework calls your code components when needed, rather than your code explicitly calling framework components.
- **Benefits:** Reduces coupling, improves modularity, and simplifies testing.

Dependency Injection (DI)

Dependency Injection is a specific pattern used to implement Inversion of Control. It's the process of providing external dependencies to a software component instead of the component creating them itself.

- **How it Works:** The Spring IoC container creates objects (beans), configures them, and "injects" their dependencies at runtime.
- **Benefits:**
  - **Loose Coupling:** Components are not responsible for creating or managing their dependencies.
  - **Testability:** Dependencies can be easily mocked or stubbed for unit testing.
  - **Maintainability:** Easier to change implementations of dependencies without affecting the dependent code.

Annotation-Based Configuration

Modern Spring development heavily relies on annotations for configuration, greatly reducing the need for verbose XML.

- `@Configuration`: Marks a class as a source of bean definitions.
- `@Bean`: Marks a method that produces a bean to be managed by the Spring container.
- `@Component`: A generic stereotype for any Spring-managed component.
- `@Autowired`: Automatically injects dependencies.

Spring Core Container

The Spring Core Container is the fundamental module that provides the IoC container. It manages the lifecycle of your application's objects (beans).

- **`BeanFactory`:** The simplest IoC container, providing basic DI functionality.
- **`ApplicationContext`:** An advanced form of `BeanFactory` that provides more enterprise-specific features like AOP integration, event publishing, message source handling, and web application capabilities. It's the preferred choice for most applications.

Setter-Based Dependency Injection

In this approach, the container injects dependencies into a bean's properties by calling public setter methods after the bean has been instantiated.

Example:

```
// Dependency
public class EmailService {
    public void sendEmail(String to, String message) {
        System.out.println("Sending email to " + to + ": " + message);
    }
}

// Dependent class
public class UserService {
    private EmailService emailService;

    // Setter method for dependency injection
    public void setEmailService(EmailService emailService) {
        this.emailService = emailService;
    }

    public void registerUser(String username, String email) {
        // ... business logic ...
        emailService.sendEmail(email, "Welcome, " + username + "!");
    }
}

// Spring Configuration
@Configuration
public class AppConfig {
    @Bean
    public EmailService emailService() {
        return new EmailService();
    }

    @Bean
    public UserService userService() {
        UserService userService = new UserService();
        userService.setEmailService(emailService()); // Manual wiring
        return userService;
    }
}
```

Constructor-Based DI

Dependencies are provided as arguments to the bean's constructor. This ensures that the bean is created in a valid state with all its mandatory dependencies. It is generally preferred for mandatory dependencies as it promotes immutability.

Example:

```
public class UserService {
    private final EmailService emailService; // Make it final for immutability

    // Constructor for dependency injection
    public UserService(EmailService emailService) {
        this.emailService = emailService;
    }

    public void registerUser(String username, String email) {
        // ... business logic ...
        emailService.sendEmail(email, "Welcome, " + username + "!");
    }
}

// Spring Configuration
@Configuration
public class AppConfig {
    @Bean
    public EmailService emailService() {
        return new EmailService();
    }

    @Bean
    public UserService userService(EmailService emailService) { // Spring
automatically injects EmailService
        return new UserService(emailService);
    }
}
```

Factory Method-Based DI

Dependencies are provided by a static or instance factory method. This approach is less common for typical application components but useful when externalizing object creation logic.

Example (less common in modern Spring unless dealing with external libraries):

```
public class Printer {
    private String type;
    private Printer(String type) { this.type = type; } // Private constructor

    public static Printer createInkjetPrinter() { // Static factory method
        return new Printer("Inkjet");
    }

    public static Printer createLaserPrinter() { // Static factory method
        return new Printer("Laser");
    }
    // ...
}
```

```
// Spring Configuration
@Configuration
public class AppConfig {
    @Bean
    public Printer inkjetPrinter() {
        return Printer.createInkjetPrinter(); // Calling static factory method
    }
}
```

Creating Spring Java SE Application in STS

While specific STS steps are visual, the conceptual process involves:

1. **Create a New Maven Project:** In STS (Spring Tool Suite), go to `File > New > Maven Project`.
2. **Add Spring Core Dependencies:** In the `pom.xml`, add `spring-context` dependency.
3. **Create Spring Configuration:** Create a Java class annotated with `@Configuration` to define your beans.
4. **Create Beans:** Write your POJOs (e.g., `UserService`, `EmailService`).
5. **Main Application Class:** In your `main` method, instantiate `AnnotationConfigApplicationContext` with your `@Configuration` class, then `getBean()` to retrieve and use your configured beans.

Example Main Class:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MySpringApp {
    public static void main(String[] args) {
        // Load Spring context from annotation configuration
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

        // Retrieve a bean from the context
        UserService userService = context.getBean(UserService.class);

        // Use the bean
        userService.registerUser("john.doe", "john@example.com");

        // Close the context (important for non-web apps)
        ((AnnotationConfigApplicationContext) context).close();
    }
}
```

Scopes of Spring Beans

A Spring bean's scope defines the lifecycle and visibility of a bean within the Spring IoC container. It determines how many instances of a bean are created and how they are managed.

- `singleton` **(Default):**
  - **Description:** Only one instance of the bean is created per Spring IoC container. This single instance is shared and returned for all subsequent requests for that bean.
  - **Use Case:** Stateless services, DAOs, utility classes, and most application components.
  - **Configuration:** `@Scope("singleton")` or simply omit `@Scope` as it's the default.
- `prototype`**:**
  - **Description:** A new instance of the bean is created every time it is requested from the Spring IoC container.
  - **Use Case:** Statefule beans where each client needs its own independent instance (e.g., a shopping cart, a user-specific configuration).
  - **Configuration:** `@Scope("prototype")`
- **Web-aware Scopes (only for web applications):**
  - `request`**:** A new instance of the bean is created for each HTTP request. The bean instance is valid only for the duration of that request.
    - **Configuration:** `@RequestScope` (or `@Scope("request")`)
  - `session`**:** A new instance of the bean is created for each HTTP session. The bean instance is valid for the duration of that session.
    - **Configuration:** `@SessionScope` (or `@Scope("session")`)
  - `application`**:** A single instance of the bean is created per `ServletContext` (web application). This is similar to a singleton but tied to the web application's lifecycle.
    - **Configuration:** `@ApplicationScope` (or `@Scope("application")`)
  - `websocket`**:** (Spring 4.0+) A new instance of the bean is created for each WebSocket session.
    - **Configuration:** `@Scope("websocket")`

Example:

```
@Component
@Scope("prototype") // Each time this bean is requested, a new instance is created
public class ShoppingCart {
    private List<String> items = new ArrayList<>();

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

    public List<String> getItems() {
        return items;
    }
}

@Component
@Scope("singleton") // Default, only one instance shared across the app
public class ProductCatalog {
    public List<String> getAllProducts() {
        return Arrays.asList("Laptop", "Mouse", "Keyboard");
    }
}
```

Spring Beans Lifecycle

The lifecycle of a Spring bean refers to the sequence of events that occur from the time a bean is instantiated by the Spring container until it is destroyed. Understanding the lifecycle allows you to perform initialization and destruction logic at appropriate times.

Key Phases/Callbacks:

1. **Instantiation:** The container instantiates the bean (calls its constructor).
2. **Populate Properties:** Dependencies (properties) are set via setters or autowiring.
3. `BeanNameAware` **callback (if implemented):** The bean's `setBeanName()` method is called, passing its ID.
4. `BeanFactoryAware` **callback (if implemented):** The bean's `setBeanFactory()` method is called, passing the `BeanFactory` that created it.
5. `ApplicationContextAware` **callback (if implemented):** The bean's `setApplicationContext()` method is called, passing the `ApplicationContext` that created it. (If using `ApplicationContext`)
6. `BeanPostProcessors` **(pre-initialization):**
    - `postProcessBeforeInitialization()` method of any registered `BeanPostProcessor` is called.
    - This allows modification of the bean instance before any initialization callbacks.
7. `@PostConstruct` **(or** `InitializingBean.afterPropertiesSet()` **or custom** `init-method`**):**
    - `@PostConstruct` **(JSR-250 annotation):** A method annotated with `@PostConstruct` is called. **(Recommended)**
    - `InitializingBean` **interface:** If the bean implements `InitializingBean`, its `afterPropertiesSet()` method is called.
    - **Custom** `init-method`**:** A method specified in the `@Bean` annotation's `initMethod` attribute or XML configuration.
    - These methods are for custom initialization logic (e.g., populating caches, opening connections).
8. `BeanPostProcessors` **(post-initialization):**
    - `postProcessAfterInitialization()` method of any registered `BeanPostProcessor` is called.
    - This allows further modification or wrapping of the bean instance (e.g., for AOP proxying).
9. **Bean is Ready:** The bean is now fully initialized and ready for use.
10. **Container Shutdown:** When the Spring IoC container is shut down.
11. `@PreDestroy` **(or** `DisposableBean.destroy()` **or custom** `destroy-method`**):**
    - `@PreDestroy` **(JSR-250 annotation):** A method annotated with `@PreDestroy` is called. **(Recommended)**
    - `DisposableBean` **interface:** If the bean implements `DisposableBean`, its `destroy()` method is called.
    - **Custom** `destroy-method`**:** A method specified in the `@Bean` annotation's `destroyMethod` attribute or XML configuration.
    - These methods are for custom cleanup logic (e.g., closing connections, releasing resources).

Example using `@PostConstruct` **and** `@PreDestroy`**:**

```
import org.springframework.stereotype.Component;
import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;
```

```
@Component
public class DatabaseConnector {

    private String connectionUrl;

    public DatabaseConnector() {
        System.out.println("1. DatabaseConnector: Constructor called.");
    }

    // This method will be called after dependency injection and initialization
    @PostConstruct
    public void init() {
        this.connectionUrl = "jdbc:mysql://localhost:3306/mydb";
        System.out.println("2. DatabaseConnector: @PostConstruct - Initializing
connection to " + connectionUrl);
        // Simulate opening a connection
    }

    public void fetchData() {
        System.out.println("3. DatabaseConnector: Fetching data using " +
connectionUrl);
    }

    // This method will be called just before the bean is destroyed
    @PreDestroy
    public void destroy() {
        System.out.println("4. DatabaseConnector: @PreDestroy - Closing
connection.");
        // Simulate closing the connection
    }
}
```

Autowiring vs Explicit Wiring

These terms refer to different ways of managing and injecting dependencies into Spring beans.

- **Explicit Wiring (Manual Wiring):**
    - **Description:** You explicitly define the relationships between beans, telling Spring exactly which bean to inject into another. This is done either through Java `@Configuration` classes by calling setter/constructor methods directly or through XML configuration.
    - **Control:** Offers fine-grained control over how dependencies are resolved.
    - **Clarity:** The wiring is explicitly visible in the configuration.
    - **Downside:** Can be verbose and tedious for large numbers of dependencies or when dependencies change frequently.

  Example (Java Config):

```
@Configuration
public class AppConfig {
    @Bean
    public ServiceA serviceA() {
        return new ServiceA();
    }

    @Bean
    public ServiceB serviceB() {
        ServiceB serviceB = new ServiceB();
        serviceB.setServiceA(serviceA()); // Explicitly telling Spring to
inject serviceA
        return serviceB;
    }
}
```

- **Autowiring:**
    - **Description:** Spring automatically resolves and injects dependencies into beans based on certain criteria (by type, by name). You annotate the dependency declaration in your class, and Spring does the rest.
    - **Convention over Configuration:** Reduces boilerplate and configuration.
    - **Speed:** Speeds up development, especially for large projects with many dependencies.
    - **Downside:** Can sometimes lead to ambiguity if multiple beans of the same type exist (requires `@Qualifier`). Less explicit than manual wiring.

    Example (Annotation Config):

```
@Component
public class ServiceA { /* ... */ }

@Component
public class ServiceB {
    @Autowired // Spring will find a ServiceA bean and inject it here
    private ServiceA serviceA;
    /* ... */
}
```

Autowiring using Annotation Configuration

The `@Autowired` annotation is the primary way to perform autowiring in modern Spring applications. Spring scans your components and automatically wires the matching dependencies.

- **`@Autowired`:**
    - Can be applied to fields, setter methods, and constructors.
    - By default, it performs autowiring by type. If multiple beans of the same type are found, Spring throws `NoUniqueBeanDefinitionException`.

- If no matching bean is found, Spring throws `NoSuchBeanDefinitionException`. This can be made optional by setting `required=false` (`@Autowired(required=false)`).

- **`@Qualifier`:**
  - Used in conjunction with `@Autowired` when there are multiple beans of the same type, to specify which particular bean to inject by name.

  Example:

```
@Component("emailServicePrimary")
public class PrimaryEmailService implements EmailService { /* ... */ }

@Component("emailServiceSecondary")
public class SecondaryEmailService implements EmailService { /* ... */ }

@Component
public class NotificationService {
    @Autowired
    @Qualifier("emailServicePrimary") // Specify which EmailService to
inject
    private EmailService emailService;
    /* ... */
}
```

Field/Setter Level Autowiring

- **Field-level Autowiring:**
  - **Description:** The `@Autowired` annotation is placed directly on the field. Spring injects the dependency directly into the field using reflection.
  - **Pros:** Concise, less boilerplate code.
  - **Cons:** Makes the class harder to unit test without Spring, as dependencies cannot be easily set externally. Breaks immutability if fields are `final`.
  - **Example:**

```
@Component
public class MyService {
    @Autowired
    private DependencyA dependencyA;
    // ...
}
```

- **Setter-level Autowiring:**
  - **Description:** The `@Autowired` annotation is placed on the setter method of the dependency. Spring calls the setter method to inject the dependency.

- **Pros:** Allows the class to be instantiated and dependencies injected using traditional constructor injection or manual setters for testing outside Spring.
- **Cons:** Requires a setter method for each dependency.
- **Example:**

```
@Component
public class MyService {
    private DependencyB dependencyB;

    @Autowired
    public void setDependencyB(DependencyB dependencyB) {
        this.dependencyB = dependencyB;
    }
    // ...
}
```

Constructor Autowiring

- **Description:** The `@Autowired` annotation is placed on the constructor. Spring uses this constructor to instantiate the bean and inject all its arguments as dependencies. If there's only one constructor, `@Autowired` is optional (Spring automatically uses it).
- **Pros:**
  - **Mandatory Dependencies:** Ensures that the bean is always created with all its necessary dependencies, preventing `NullPointerException`s.
  - **Immutability:** Supports immutable components (by making fields `final`), which can lead to safer and more predictable code.
  - **Testability:** Makes components easier to unit test outside of the Spring container as dependencies can be passed directly to the constructor.
- **Cons:** Can become verbose for classes with many dependencies.
- **Recommendation:** Generally the **preferred** method for injecting mandatory dependencies.
- **Example:**

```
@Component
public class MyService {
    private final DependencyC dependencyC;

    @Autowired // Optional if there's only one constructor
    public MyService(DependencyC dependencyC) {
        this.dependencyC = dependencyC;
    }
    // ...
}
```

Introduction to Spring Boot

Spring Boot is an extension of the Spring Framework that aims to simplify the development of production-ready Spring applications. It emphasizes "convention over configuration" to reduce boilerplate setup and accelerate development.

- **Opinionated Defaults:** Spring Boot provides sensible default configurations for common Spring features, libraries, and external tools.
- **Auto-configuration:** Automatically configures your Spring application based on the JARs present on your classpath. For example, if you add `spring-boot-starter-web`, Spring Boot automatically configures DispatcherServlet and other web-related components.
- **Embedded Servers:** Can embed Tomcat, Jetty, or Undertow directly into a JAR file, allowing you to run applications as standalone executables without needing to deploy to a separate web server.
- **Starter POMs:** Provide convenient dependency descriptors to simplify Maven/Gradle build configurations. A starter bundles common dependencies for a particular feature (e.g., `spring-boot-starter-web`, `spring-boot-starter-data-jpa`).
- **Actuator:** Provides production-ready features like monitoring, health checks, and metrics.

Hello Spring Boot Java SE Application & Its Execution

A minimal Spring Boot application can be a simple Java SE application that leverages Spring Boot's auto-configuration.

Example `main` class:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication // Combines @Configuration, @EnableAutoConfiguration,
@ComponentScan
public class MySpringBootApp {

    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run(MySpringBootApp.class,
args);

        // You can retrieve beans from the context just like a regular Spring app
        // MyService myService = context.getBean(MyService.class);
        // myService.doSomething();

        System.out.println("Spring Boot application started successfully!");
    }
}
```

Execution:

1. **Maven/Gradle Build:**

    - For Maven, ensure you have the `spring-boot-maven-plugin` in your `pom.xml`.
    - Run `mvn clean install` to build the JAR.
2. **Run as Executable JAR:**
    - Navigate to your `target` directory (for Maven) or `build/libs` (for Gradle).
    - Run `java -jar your-app-name.jar`.
    - If it's a web application, the embedded server (e.g., Tomcat) will start on its default port (8080).

Stereo-type Annotations

Stereotype annotations are specializations of `@Component` that indicate the role of a component within a Spring application. Spring's component scanning feature automatically detects and registers beans annotated with these.

- `@Component`:
    - **Purpose:** A generic stereotype for any Spring-managed component. It indicates that a class is a Spring component and should be registered as a bean in the Spring container.
    - **Use Case:** General-purpose beans, utility classes.
- `@Service`:
    - **Purpose:** Specialization of `@Component` for service layer classes. It conveys the semantic meaning that the class holds business logic.
    - **Benefits:** Can be targets for AOP (e.g., transaction management), and potentially provides better separation of concerns.
    - **Use Case:** Classes implementing business logic, often delegating to DAO/Repository layer.
- `@Repository`:
    - **Purpose:** Specialization of `@Component` for data access layer (DAO) classes.
    - **Benefits:** Enables automatic exception translation for persistence technologies (e.g., converting JDBC/Hibernate-specific exceptions to Spring's `DataAccessException` hierarchy).
    - **Use Case:** Classes interacting directly with a database or other persistence mechanisms.
- `@Controller`:
    - **Purpose:** Specialization of `@Component` for Spring MVC controller classes. It handles incoming web requests and prepares a model to be rendered by a view.
    - **Use Case:** Building traditional web applications that serve HTML views.
- `@RestController`:
    - **Purpose:** A convenience annotation that combines `@Controller` and `@ResponseBody`. It indicates that the class is a controller where all methods return domain objects directly (or JSON/XML), rather than a view name.
    - **Use Case:** Building RESTful web services that return data directly to the client (e.g., JSON).

Relationship: `@Service`, `@Repository`, `@Controller`, and `@RestController` are all meta-annotated with `@Component`. This means that if you component-scan for `@Component`, these specialized annotations will also be picked up.

ORM Concept

- **ORM (Object-Relational Mapping):** A programming technique that maps objects in an object-oriented programming language (like Java) to tables in a relational database.

- **Problem Solved:** Bridges the "object-relational impedance mismatch" – the differences in how data is represented in object-oriented programming (objects, inheritance, polymorphism) and relational databases (tables, rows, columns, foreign keys).
- **How it Works:** An ORM tool automatically handles the mapping between object properties and database columns, and translates object-oriented operations (creating objects, updating properties, deleting objects) into SQL queries.
- **Benefits:**
  - **Productivity:** Developers work with objects, reducing the need to write repetitive SQL.
  - **Maintainability:** Code is cleaner and easier to understand.
  - **Database Agnostic:** Applications can potentially work with different database systems by changing configuration, as the ORM handles dialect-specific SQL.
  - **Portability:** ORM implementations often handle vendor-specific SQL nuances.

JPA

- **JPA (Java Persistence API):** A Java specification for managing relational data in applications. It defines a standard API for ORM.
- **Specification vs. Implementation:** JPA is a *specification* (a set of interfaces and annotations), not a direct implementation. It provides a blueprint for how ORM should work in Java.
- **Purpose:** To simplify persistence logic in Java applications and provide a standardized way to interact with databases using ORM.
- **Providers:** Popular implementations of the JPA specification include Hibernate (the most widely used), EclipseLink, OpenJPA, etc.

JPA ORM Annotations

JPA uses annotations to define the mapping between Java objects (entities) and database tables.

- `@Entity`**:** Marks a plain Java class as a JPA entity, meaning it corresponds to a table in the database.

```
@Entity
public class Product { /* ... */ }
```

- `@Table(name="...")`**:** (Optional) Specifies the name of the database table if it differs from the entity class name.

```
@Entity
@Table(name = "products_tbl")
public class Product { /* ... */ }
```

- `@Id`**:** Marks a field as the primary key of the entity.

```
@Id
private Long id;
```

- `@GeneratedValue`: Specifies how the primary key value is generated.
  - `strategy = GenerationType.AUTO`: (Default) Lets the persistence provider choose the best strategy for the database.
  - `strategy = GenerationType.IDENTITY`: Uses an auto-incrementing column (e.g., MySQL `AUTO_INCREMENT`, SQL Server `IDENTITY`).
  - `strategy = GenerationType.SEQUENCE`: Uses a database sequence (e.g., Oracle). Requires `@SequenceGenerator`.
  - `strategy = GenerationType.TABLE`: Uses a table to simulate a sequence. Requires `@TableGenerator`.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

- `@Column(name="...", length=..., nullable=...)`: (Optional) Specifies details of the database column that a field maps to.
  - `name`: Column name if different from field name.
  - `length`: Max length for string columns.
  - `nullable`: If the column can be null.
  - `unique`: If the column values must be unique.

```
@Column(name = "product_name", length = 255, nullable = false, unique =
true)
private String name;
```

- `@Transient`: Marks a field that should *not* be persisted to the database. It exists only in the Java object.

```
@Transient
private int temporaryValue;
```

- **Relationships:** `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany` (discussed later).

Introduction to Spring Data

Spring Data is a powerful project within the Spring ecosystem that aims to simplify data access layers significantly. It provides a consistent programming model for various data stores while still maintaining their specific characteristics.

- **Repository Abstraction:** Its core concept is the `Repository` interface. You define interfaces for your data access methods, and Spring Data automatically generates the implementation at runtime.
- **Reduced Boilerplate:** Eliminates much of the boilerplate code traditionally required for implementing DAOs (Data Access Objects).
- **Common Functionality:** Provides common CRUD (Create, Read, Update, Delete) operations out of the box.

- **Query Methods:** Allows you to define query methods by simply declaring method names that follow a specific naming convention (e.g., `findByLastNameAndFirstName()`). Spring Data parses these names and generates the corresponding queries.
- **Support for Multiple Data Stores:** Offers sub-projects for various databases: Spring Data JPA (for relational databases), Spring Data MongoDB, Spring Data Redis, Spring Data Elasticsearch, etc.

Spring Data Architecture

Spring Data's architecture is built around the `Repository` abstraction, which dramatically simplifies data access code.

- **Repository Interfaces:** At the core are interfaces that you define. These interfaces extend one of Spring Data's base interfaces:
  - **`Repository<T, ID>`:** The marker interface for all repositories. Provides no methods but allows Spring Data to detect your repository.
  - **`CrudRepository<T, ID>`:** Extends `Repository`. Provides basic CRUD operations: `save()`, `findById()`, `findAll()`, `delete()`, `count()`, etc.
  - **`PagingAndSortingRepository<T, ID>`:** Extends `CrudRepository`. Adds methods for pagination and sorting (`findAll(Pageable pageable)`, `findAll(Sort sort)`).
  - **`JpaRepository<T, ID>`:** Extends `PagingAndSortingRepository` (and `QueryByExampleExecutor`). Specifically for JPA-based data stores. Adds JPA-specific methods like `flush()`, `saveAndFlush()`, `deleteInBatch()`. This is the most commonly used interface for relational databases with JPA.
- **Custom Query Methods:** You can define custom query methods in your repository interfaces by adhering to specific naming conventions. Spring Data parses these method names and generates the corresponding queries (e.g., `findByLastNameAndFirstName(String lastName, String firstName)`).
- **Custom Implementations:** For complex queries or specific logic not covered by query methods, you can provide custom implementations for parts of your repository interface.
- **Proxy Generation:** During application startup, Spring Data dynamically generates concrete implementations of your repository interfaces using proxies. It inspects the methods defined in your interface and translates them into appropriate queries for the underlying persistence provider (e.g., Hibernate for JPA).

Example `UserRepository`:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import java.util.List;

@Repository // A stereotype annotation for repository layer
public interface UserRepository extends JpaRepository<User, Long> {
    // Custom query method based on naming convention
    List<User> findByEmailContaining(String emailPart);

    // Another custom query method
    List<User> findByAgeGreaterThan(int age);
```

```
    // Using @Query annotation for more complex queries (JPQL or native SQL)
    // @Query("SELECT u FROM User u WHERE u.firstName = ?1 AND u.lastName = ?2")
    // User findByFirstNameAndLastName(String firstName, String lastName);
}
```

Spring Data Java SE Application

While Spring Data is very popular with Spring Boot web applications, it can also be used in standalone Java SE applications. The setup involves configuring the `ApplicationContext` to scan for repositories and setting up the JPA entity manager.

Conceptual steps:

1. **Dependencies:** Add `spring-context`, `spring-data-jpa`, `hibernate-core`, `h2` (or other database driver), and `jakarta.persistence-api` to `pom.xml`.
2. **JPA Configuration:** Configure an `EntityManagerFactory` and `PlatformTransactionManager` in a `@Configuration` class.
3. **Enable JPA Repositories:** Use `@EnableJpaRepositories` annotation on your `@Configuration` class, specifying the package where your repository interfaces are located.
4. **Main Class:** Get an instance of your repository from the `ApplicationContext` and use its methods.

Transaction Management using `@Transactional`

Transaction management is crucial for data consistency in applications that interact with databases. Spring provides robust transaction management capabilities, primarily through its declarative transaction management using the `@Transactional` annotation.

- **ACID Properties:** Transactions ensure Atomicity, Consistency, Isolation, and Durability.
- **Declarative Transaction Management:** You define transaction boundaries (where a transaction starts and ends) using annotations or XML configuration, rather than writing explicit transaction code. Spring's AOP (Aspect-Oriented Programming) intercepts method calls and applies transactional behavior.
- `@Transactional` **Annotation:** Applied to classes or methods.
    - When applied to a class, all public methods of that class will run in a transaction.
    - When applied to a method, only that method will run in a transaction. Method-level annotations override class-level ones.
    - **Rollback:** By default, a transaction will roll back on unchecked exceptions (runtime exceptions) and commit on checked exceptions. This behavior can be customized (`rollbackFor`, `noRollbackFor`).
    - **Propagation:** Defines how transactions interact when multiple methods with `@Transactional` are called within each other (e.g., `REQUIRED`, `REQUIRES_NEW`, `SUPPORTS`). `REQUIRED` is the default: if a transaction already exists, participate in it; otherwise, create a new one.
    - **Isolation:** Defines the degree to which one transaction's changes are visible to other concurrent transactions (e.g., `READ_COMMITTED`, `SERIALIZABLE`). `DEFAULT` typically uses the database's default.
    - **Read-only:** `readOnly = true` for read-only operations, can offer performance benefits by allowing the database to perform optimizations.

Example:

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service // Mark as service layer component
public class OrderService {

    private final ProductRepository productRepository;
    private final OrderRepository orderRepository;

    public OrderService(ProductRepository productRepository, OrderRepository
orderRepository) {
        this.productRepository = productRepository;
        this.orderRepository = orderRepository;
    }

    @Transactional // This entire method runs within a single database transaction
    public Order placeOrder(Long productId, int quantity, Long userId) {
        // 1. Fetch product
        Product product = productRepository.findById(productId)
            .orElseThrow(() -> new IllegalArgumentException("Product not found"));

        // 2. Check stock and update (business logic)
        if (product.getStock() < quantity) {
            throw new RuntimeException("Insufficient stock for product: " +
product.getName());
        }
        product.setStock(product.getStock() - quantity);
        productRepository.save(product); // Update product in DB

        // 3. Create order
        Order order = new Order();
        order.setProductId(productId);
        order.setQuantity(quantity);
        order.setUserId(userId);
        order.setOrderDate(LocalDateTime.now());
        order.setStatus("PLACED");

        // 4. Save order
        return orderRepository.save(order); // Save order in DB
        // If any error occurs after product.save() and before
orderRepository.save(),
        // the entire transaction (including product update) will be rolled back.
    }

    @Transactional(readOnly = true) // This method is read-only
    public List<Order> getUserOrders(Long userId) {
        return orderRepository.findByUserId(userId);
    }
}
```

Spring Repository (DAO) Layer

The Repository layer (or Data Access Object - DAO layer) is responsible for interacting directly with the database. In Spring Data JPA, this is typically represented by interfaces that extend `JpaRepository` (or other `Repository` interfaces).

- **Role:**
  - Abstracts persistence logic from the business logic.
  - Performs CRUD operations on entities.
  - Maps database results to Java objects and vice-versa.
  - Handles database-specific concerns (though Spring Data handles most of this).
- **Implementation:** You define an interface, and Spring Data provides the implementation at runtime.
- **Example (from `Spring Data Architecture` section):**

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // ...
}
```

Spring Service Layer Implementation

The Service layer acts as an intermediary between the Controller layer (or UI layer in a Java SE app) and the Repository layer. It encapsulates the application's business logic and orchestrates operations across multiple repositories.

- **Role:**
  - Contains the core business rules and logic of the application.
  - Orchestrates calls to multiple repositories to complete a business transaction.
  - Acts as the transactional boundary (where `@Transactional` is typically applied).
  - Performs data validation and transformation related to business rules.
- **Implementation:** Typically a class annotated with `@Service`.
- **Example (from `Transaction Management` section):**

```
@Service
public class OrderService {
    // ... business methods with @Transactional ...
}
```

Spring Managed Service Layer to Handle Transactions

As seen above, the `@Transactional` annotation is most effectively used on the Service layer methods. Spring's AOP (Aspect-Oriented Programming) capabilities intercept calls to these service methods.

- **How it Works:**
  1. When a client calls a method on a `@Service` bean that is annotated with `@Transactional`, Spring's AOP proxy intercepts the call.

2. Before the method execution, Spring opens a new database transaction (or joins an existing one based on propagation settings).
3. The service method executes, potentially making multiple calls to different repository methods (`save`, `delete`, `findById`). All these operations happen within the same transaction.
4. If the method completes successfully, Spring commits the transaction.
5. If an unchecked exception (runtime exception) is thrown within the method, Spring automatically rolls back the entire transaction, ensuring data consistency.

- **Benefits:** Separates transaction management concerns from business logic, making code cleaner and more modular.

JPA ID Generators

JPA provides strategies for automatically generating primary key values for entities. This is configured using the `@GeneratedValue` annotation with the `strategy` attribute.

- `GenerationType.AUTO` **(Default):**
  - **Description:** The persistence provider (e.g., Hibernate) chooses the appropriate strategy based on the underlying database. It might use `IDENTITY`, `SEQUENCE`, or `TABLE`. This is convenient but can be less predictable across different databases.
  - **Use Case:** Simple applications or when database portability is not a strict requirement for ID generation.
- `GenerationType.IDENTITY`**:**
  - **Description:** The database column itself auto-increments. The value is generated by the database when the record is inserted. This means the ID is only available after the `persist()` (or `save()`) operation.
  - **Database Support:** Works with databases like MySQL (`AUTO_INCREMENT`), SQL Server (`IDENTITY`).
  - **Use Case:** When you want the database to handle primary key generation and don't need the ID before `persist()`.
- `GenerationType.SEQUENCE`**:**
  - **Description:** Uses a database sequence object to generate unique IDs. The persistence provider increments the sequence before inserting the record, so the ID is available immediately.
  - **Database Support:** Works with databases like Oracle, PostgreSQL.
  - **Configuration:** Often requires the `@SequenceGenerator` annotation to specify the sequence name, allocation size, etc.
  - **Use Case:** When you need the ID before committing the transaction or when using databases that support sequences.
- `GenerationType.TABLE`**:**
  - **Description:** Simulates a sequence using a dedicated database table to store and manage the next available ID.
  - **Database Support:** Database-agnostic.
  - **Configuration:** Requires the `@TableGenerator` annotation to specify the table name, primary key column, value column, etc.
  - **Use Case:** When you need sequence-like behavior but your database does not support sequences, or for database portability without relying on `AUTO`.

Example (`GenerationType.SEQUENCE`):

```
import jakarta.persistence.*;

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"user_seq_gen")
    @SequenceGenerator(name = "user_seq_gen", sequenceName = "user_sequence",
allocationSize = 1)
    private Long id;

    // ... other fields
}
```

Spring Data Query Methods

Spring Data JPA allows you to define complex queries simply by declaring method signatures in your repository interfaces, following specific naming conventions. Spring Data parses these method names and generates the corresponding JPA queries at runtime.

- **Basic CRUD & Paging/Sorting:** Provided by `JpaRepository` directly (e.g., `save`, `findById`, `findAll`, `deleteById`, `findAll(Pageable)`).
- **Derived Query Methods (Method Name Conventions):**
  - Start with `find`, `read`, `get`, `query`, `count`, `delete`, `stream`.
  - Followed by the entity property name(s) and keywords.
  - **Keywords:**
    - `And`, `Or`: Combine conditions (`findByLastNameAndFirstName`).
    - `Between`: For ranges (`findByStartDateBetween`).
    - `LessThan`, `GreaterThan`: For comparisons (`findByAgeGreaterThan`).
    - `IsNotNull`, `IsNull`: For null checks (`findByEmailIsNotNull`).
    - `Like`, `Containing`, `StartingWith`, `EndingWith`: For string matching (`findByNameContaining`, `findByEmailStartingWith`).
    - `OrderBy`: For sorting (`findByAgeOrderByLastNameDesc`).
    - `Not`: Negates a condition (`findByNameNot`).
    - `In`, `NotIn`: For lists of values (`findByStatusIn`).
    - `Top`, `First`: Limiting results (`findTop3ByAge`).
  - **Parameters:** Method parameters must match the properties used in the method name.
  - **Return Types:** Can be a single entity, `Optional<Entity>`, `List<Entity>`, `Set<Entity>`, `Page<Entity>`, `Slice<Entity>`, `Stream<Entity>`, `Long`, `Integer` (for count).

Example `UserRepository` with Query Methods:

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import java.util.List;
import java.util.Optional;
```

```java
public interface UserRepository extends JpaRepository<User, Long> {

    // Find users by last name
    List<User> findByLastName(String lastName);

    // Find a user by email, returns Optional to handle absence
    Optional<User> findByEmail(String email);

    // Find users by first name and last name
    List<User> findByFirstNameAndLastName(String firstName, String lastName);

    // Find users whose email contains a specific string (case-insensitive)
    List<User> findByEmailContainingIgnoreCase(String emailPart);

    // Find users older than a certain age, ordered by last name descending
    List<User> findByAgeGreaterThanOrderByLastNameDesc(int age);

    // Find top 5 users by age
    List<User> findTop5ByOrderByAgeDesc();

    // Find users by status within a list of statuses, with pagination
    Page<User> findByStatusIn(List<String> statuses, Pageable pageable);

    // Count users by email
    long countByEmail(String email);

    // Delete users by active status
    long deleteByActive(boolean active);
}
```

Spring Entity Relations

In object-relational mapping, establishing relationships between entities is crucial to model real-world data structures accurately. JPA provides annotations to define these relationships.

- **Cardinality:**
    - **One-to-One (`@OneToOne`):** Each instance of Entity A is associated with at most one instance of Entity B, and vice-versa.
    - **One-to-Many (`@OneToMany`):** Each instance of Entity A can be associated with multiple instances of Entity B, but each instance of Entity B is associated with at most one instance of Entity A.
    - **Many-to-One (`@ManyToOne`):** Each instance of Entity B can be associated with multiple instances of Entity A, but each instance of Entity A is associated with at most one instance of Entity B. (This is the inverse of One-to-Many).
    - **Many-to-Many (`@ManyToMany`):** Each instance of Entity A can be associated with multiple instances of Entity B, and each instance of Entity B can be associated with multiple instances of Entity A.
- `mappedBy` **attribute:** Used on the "owning" side of a bidirectional relationship. It refers to the field in the inverse (non-owning) side of the relationship. This tells JPA that the foreign key column is defined in

the inverse entity.

- `FetchType` **attribute:** Determines when the associated entities are loaded.
    - `FetchType.LAZY` **(Default for OneToMany/ManyToMany):** The associated entities are loaded only when they are explicitly accessed. Improves performance by avoiding loading unnecessary data.
    - `FetchType.EAGER` **(Default for OneToOne/ManyToOne):** The associated entities are loaded immediately along with the owning entity. Can lead to performance issues if used indiscriminately (N+1 query problem).
- `CascadeType` **attribute:** Defines how persistence operations (persist, merge, remove, refresh, detach) are cascaded from the parent entity to its associated child entities.
    - `CascadeType.ALL`: Cascades all operations.
    - `CascadeType.PERSIST`: Cascades `persist` operations.
    - `CascadeType.MERGE`: Cascades `merge` operations.
    - `CascadeType.REMOVE`: Cascades `remove` operations.
    - `CascadeType.REFRESH`: Cascades `refresh` operations.
    - `CascadeType.DETACH`: Cascades `detach` operations.

One-to-One (`@OneToOne`)

A single entity instance is related to a single other entity instance.

Example: `User` **and** `UserProfile`

```
// User.java (Owning side)
@Entity
public class User {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;

    @OneToOne(cascade = CascadeType.ALL) // User owns the relationship, cascades
all operations
    @JoinColumn(name = "profile_id", referencedColumnName = "id") // Defines the
foreign key column
    private UserProfile profile;

    // Getters and Setters
}

// UserProfile.java (Non-owning side, mappedBy refers to the field in User)
@Entity
public class UserProfile {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String bio;
    private String phoneNumber;
```

```
    @OneToOne(mappedBy = "profile") // 'profile' is the field name in the User
entity
    private User user;

    // Getters and Setters
}
```

One-to-Many (`@OneToMany`)

One entity instance is related to multiple instances of another entity.

Example: `Department` and `Employee`

```
// Department.java (One-to-Many, owning side for operations)
@Entity
public class Department {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL, orphanRemoval =
true)
    // 'department' is the field name in the Employee entity
    // orphanRemoval = true: If an Employee is removed from the collection, it's
deleted from DB
    private List<Employee> employees = new ArrayList<>();

    // Helper methods to manage bidirectional relationship
    public void addEmployee(Employee employee) {
        employees.add(employee);
        employee.setDepartment(this);
    }

    public void removeEmployee(Employee employee) {
        employees.remove(employee);
        employee.setDepartment(null);
    }
    // Getters and Setters
}

// Employee.java (Many-to-One, owning side for the foreign key)
@Entity
public class Employee {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;

    @ManyToOne(fetch = FetchType.LAZY) // Default for ManyToOne is EAGER, but LAZY
is often better
    @JoinColumn(name = "department_id") // Foreign key column in Employee table
```

```
    private Department department;

    // Getters and Setters
}
```

Many-to-One (`@ManyToOne`)

Multiple instances of one entity can be related to a single instance of another entity. This is always the owning side in a `OneToMany`-`ManyToOne` relationship.

Example: `Employee` and `Department` (from above, `Employee` is `ManyToOne` to `Department`)

```
// Employee.java (Many-to-One, owning side for the foreign key)
@Entity
public class Employee {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;

    @ManyToOne(fetch = FetchType.LAZY) // Default for ManyToOne is EAGER, but LAZY
is often better
    @JoinColumn(name = "department_id") // Foreign key column in Employee table
    private Department department; // This field refers to the "one" side

    // Getters and Setters
}
```

Many-to-Many (`@ManyToMany`)

Multiple instances of one entity can be related to multiple instances of another entity. This typically requires a join table in the database.

Example: `Student` and `Course`

```
// Student.java (Owning side - typically the side that initiates the relationship
saving)
@Entity
public class Student {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE}) // Cascade
persist/merge ops
    @JoinTable(
        name = "student_course", // Name of the join table
        joinColumns = @JoinColumn(name = "student_id"), // FK column for Student
in join table
```

```
        inverseJoinColumns = @JoinColumn(name = "course_id") // FK column for
Course in join table
    )
    private Set<Course> courses = new HashSet<>();

    // Helper method to maintain relationship consistency
    public void addCourse(Course course) {
        this.courses.add(course);
        course.getStudents().add(this);
    }

    public void removeCourse(Course course) {
        this.courses.remove(course);
        course.getStudents().remove(this);
    }
    // Getters and Setters
}

// Course.java (Non-owning side - mappedBy refers to the field in Student)
@Entity
public class Course {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;

    @ManyToMany(mappedBy = "courses") // 'courses' is the field name in the
Student entity
    private Set<Student> students = new HashSet<>();

    // Getters and Setters
}
```

JPQL Queries

JPQL (Java Persistence Query Language): An object-oriented query language used to perform queries against entities stored in a relational database. It is similar in syntax to SQL but operates on entity objects and their attributes rather than database tables and columns.

- **Purpose:** To define flexible queries that are independent of the underlying database schema.
- **Syntax:**
  - Uses entity names (e.g., `User`, `Product`) and entity attributes (e.g., `user.firstName`) instead of table and column names.
  - Supports `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`.
  - Supports aggregate functions (`COUNT`, `SUM`, `AVG`, `MAX`, `MIN`).
  - Supports joins.
- **Execution:** Executed via `EntityManager.createQuery()` or through `@Query` annotation in Spring Data JPA repositories.

Example `UserRepository` with `@Query` (JPQL):

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import java.util.List;

public interface UserRepository extends JpaRepository<User, Long> {

    // Find users by last name using JPQL
    @Query("SELECT u FROM User u WHERE u.lastName = ?1") // Positional parameter
    List<User> findByLastNameJPQL(String lastName);

    // Find users by first name and last name using named parameters
    @Query("SELECT u FROM User u WHERE u.firstName = :firstName AND u.lastName =
:lastName")
    List<User> findByFirstNameAndLastNameJPQL(@Param("firstName") String
firstName, @Param("lastName") String lastName);

    // Find users older than a certain age using JPQL
    @Query("SELECT u FROM User u WHERE u.age > :age ORDER BY u.lastName DESC")
    List<User> findUsersOlderThanJPQL(@Param("age") int age);

    // Counting users by status
    @Query("SELECT COUNT(u) FROM User u WHERE u.status = :status")
    long countUsersByStatusJPQL(@Param("status") String status);
}
```

Named Queries

Named Queries are JPQL or native SQL queries that are defined once and given a name. They are often defined in the entity class itself or in an external XML file. This improves readability, reusability, and allows for early validation by the persistence provider.

- `@NamedQuery`**:** Used to define a single named query.
- `@NamedQueries`**:** Used to group multiple named queries.

Example `User` entity with Named Query:

```
import jakarta.persistence.*;
import java.util.List;

@Entity
@Table(name = "users")
@NamedQueries({
    @NamedQuery(name = "User.findByEmailAddress",
                query = "SELECT u FROM User u WHERE u.email = :email"),
    @NamedQuery(name = "User.findAllActiveUsers",
                query = "SELECT u FROM User u WHERE u.active = true")
})
public class User {
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private int age;
    private boolean active;
    private String status;

    // Getters and Setters
}
```

Using Named Queries in Spring Data JPA:

By default, Spring Data JPA automatically checks for named queries that match the method name following the pattern `EntityName.MethodName`.

```
// UserRepository.java
public interface UserRepository extends JpaRepository<User, Long> {
    // This will automatically resolve to the named query
"User.findByEmailAddress"
    List<User> findByEmailAddress(@Param("email") String email);

    // This will automatically resolve to the named query
"User.findAllActiveUsers"
    List<User> findAllActiveUsers();
}
```

Alternatively, you can explicitly reference a named query using the `@Query` annotation:

```
// UserRepository.java
public interface UserRepository extends JpaRepository<User, Long> {
    @Query(name = "User.findByEmailAddress") // Explicitly reference the named
query
    List<User> findUserByEmail(@Param("email") String email);
}
```

Introduction to Web Services

Web services are standardized ways to integrate web-based applications using open standards (XML, SOAP, WSDL, UDDI) or more commonly, simpler approaches like REST. They enable different applications to communicate and exchange data over a network, regardless of their underlying technologies.

- **Client-Server Architecture:** Web services typically follow a client-server model, where a client sends a request to a server, and the server processes the request and sends back a response.

- **Interoperability:** The primary goal of web services is to enable interoperability between disparate systems.
- **Statelessness (for REST):** Each request from the client to the server contains all the information needed to understand the request, and the server does not store any client context between requests.

REST Architecture

REST (Representational State Transfer): An architectural style for designing networked applications. RESTful web services (RESTful APIs) are stateless, client-server, cacheable systems that use HTTP methods to operate on resources.

- **Resources:** Everything is a resource (e.g., a `User`, an `Order`, a `Product`). Resources are identified by URIs (Uniform Resource Identifiers).
  - Example: `/users`, `/products/123`, `/orders`
- **Standard HTTP Methods:** REST uses standard HTTP methods to perform actions on resources.
  - **GET:** Retrieve a resource (read-only).
    - Example: `GET /products/123`
  - **POST:** Create a new resource.
    - Example: `POST /products` (with product data in body)
  - **PUT:** Update an existing resource (replaces the entire resource).
    - Example: `PUT /products/123` (with updated product data in body)
  - **PATCH:** Partially update an existing resource.
    - Example: `PATCH /products/123` (with partial product data in body)
  - **DELETE:** Remove a resource.
    - Example: `DELETE /products/123`
- **Stateless:** Each request from the client to the server must contain all the information needed to understand the request. The server should not rely on any previous requests from the client.
- **Uniform Interface:** A consistent way to interact with resources (e.g., using standard HTTP methods).
- **Client-Server:** Clear separation of concerns between client and server.
- **Cacheable:** Responses can be cached to improve performance.
- **Layered System:** Can have intermediate servers (proxies, load balancers) without affecting client-server interaction.

`@Controller` vs `@RestController`

These are stereotype annotations for defining controllers in Spring MVC and Spring Boot.

- **`@Controller`:**
  - **Purpose:** Marks a class as a Spring MVC controller. Typically used in traditional web applications where the methods return a view name (e.g., a Thymeleaf or JSP template) to be rendered by the ViewResolver.
  - **Return Type:** Methods usually return `String` (view name) or `ModelAndView`.
  - **View Resolution:** Requires a view resolver to translate the logical view name into a physical view (e.g., `/WEB-INF/views/home.jsp`).
  - **Example:**

```
@Controller
public class WebController {
```

```
    @GetMapping("/home")
    public String showHomePage(Model model) {
        model.addAttribute("message", "Welcome!");
        return "home"; // Returns the logical view name "home"
    }
}
```

- `@RestController`:
  - **Purpose:** A convenience annotation that combines `@Controller` and `@ResponseBody`. It indicates that the class is a RESTful web service controller where all methods directly return domain objects or data (e.g., JSON, XML) to the client, rather than a view name.
  - **Return Type:** Methods return objects that Spring automatically serializes to the response body (e.g., JSON).
  - **View Resolution:** No view resolution is involved.
  - **Use Case:** Ideal for building REST APIs.
  - **Example:**

```
@RestController // Combines @Controller and @ResponseBody
@RequestMapping("/api/products")
public class ProductRestController {
    @GetMapping("/{id}")
    public Product getProductById(@PathVariable Long id) {
        // ... fetch product from service ...
        return new Product(id, "Laptop", 1200.0); // Returns a Product
object, serialized to JSON
    }
}
```

`@RequestBody`, `@ResponseBody`, `ResponseEntity`

These annotations and class are crucial for handling HTTP request and response bodies in Spring REST services.

- `@RequestBody`:
  - **Purpose:** Used as a method parameter annotation. It indicates that a method parameter should be bound to the body of the HTTP request. Spring automatically converts the incoming request body (e.g., JSON, XML) into the specified Java object using an `HttpMessageConverter`.
  - **Use Case:** For `POST` and `PUT` requests where the client sends data in the request body.
  - **Example:**

```
@PostMapping
public Product createProduct(@RequestBody Product product) { // Product
object created from request JSON
    // ... save product ...
    return product;
}
```

- `@ResponseBody`:
  - **Purpose:** Used as a method-level annotation (or implicitly included by `@RestController`). It indicates that the return value of a method should be bound to the body of the HTTP response. Spring automatically converts the Java object into a suitable format (e.g., JSON, XML) before sending it back to the client.
  - **Use Case:** For `GET`, `POST`, `PUT`, `DELETE` methods where the server sends data back in the response body.
  - **Example (if using `@Controller` instead of `@RestController`):**

```
@Controller
public class ProductController {
    @GetMapping("/{id}")
    @ResponseBody // Explicitly indicates return value should be in
response body
    public Product getProductById(@PathVariable Long id) {
        return new Product(id, "Tablet", 600.0);
    }
}
```

- `ResponseEntity`:
  - **Purpose:** A class that represents the entire HTTP response: status code, headers, and body. It gives you full control over the HTTP response.
  - **Use Case:** When you need to return specific HTTP status codes (other than 200 OK or 201 Created for POST), custom headers, or different types of bodies based on logic.
  - **Example:**

```
@GetMapping("/{id}")
public ResponseEntity<Product> getProductById(@PathVariable Long id) {
    Optional<Product> product = productService.findById(id);
    if (product.isPresent()) {
        return ResponseEntity.ok(product.get()); // Returns 200 OK with
product
    } else {
        return ResponseEntity.notFound().build(); // Returns 404 Not
Found
    }
}

@PostMapping
public ResponseEntity<Product> createProduct(@RequestBody Product
product) {
    Product savedProduct = productService.save(product);
    return
ResponseEntity.status(HttpStatus.CREATED).body(savedProduct); //
```

```
    Returns 201 Created
    }
```

Building Restful Web Services with Spring Boot

Spring Boot makes building RESTful web services straightforward using starter dependencies and annotations.

Key Annotations:

- `@RestController`: Marks the class as a REST controller.
- `@RequestMapping`: Maps HTTP requests to handler methods. Can be used at class-level (base path) or method-level (specific path).
- `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`: Specific composite annotations for common HTTP methods, combining `@RequestMapping` with the method type.
- `@PathVariable`: Extracts a variable from the URI path.
- `@RequestParam`: Extracts a query parameter from the URL.

Example REST Controller:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/api/products") // Base URL for this controller
public class ProductRestController {

    private final ProductService productService; // Injected service

    public ProductRestController(ProductService productService) {
        this.productService = productService;
    }

    // GET /api/products - Get all products
    @GetMapping
    public List<Product> getAllProducts() {
        return productService.findAll();
    }

    // GET /api/products/{id} - Get product by ID
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        Optional<Product> product = productService.findById(id);
        return product.map(ResponseEntity::ok) // If product present, return 200
OK
                      .orElseGet(() -> ResponseEntity.notFound().build()); // Else
404 Not Found
    }
```

```
    // POST /api/products - Create a new product
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED) // Returns 201 Created status
    public Product createProduct(@RequestBody Product product) {
        return productService.save(product);
    }

    // PUT /api/products/{id} - Update an existing product
    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable Long id,
@RequestBody Product productDetails) {
        return productService.findById(id)
            .map(product -> {
                product.setName(productDetails.getName());
                product.setPrice(productDetails.getPrice());
                product.setStock(productDetails.getStock());
                Product updatedProduct = productService.save(product);
                return ResponseEntity.ok(updatedProduct); // Returns 200 OK
            })
            .orElseGet(() -> ResponseEntity.notFound().build()); // Returns 404
Not Found if product not found
    }

    // DELETE /api/products/{id} - Delete a product
    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT) // Returns 204 No Content status
    public void deleteProduct(@PathVariable Long id) {
        productService.deleteById(id);
    }
}
```

Exception Handling

Proper exception handling is crucial for robust REST APIs. Spring Boot provides several ways to manage exceptions, ensuring consistent error responses.

- **`@ResponseStatus`:** Directly annotate a custom exception class with `@ResponseStatus` to return a specific HTTP status code when that exception is thrown.

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}

// In controller:
@GetMapping("/{id}")
public Product getProductById(@PathVariable Long id) {
    return productService.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Product not found
```

```
with ID: " + id));
}
```

- `@ExceptionHandler`:
  - Used on methods within a `@Controller` or `@RestController` to handle specific exceptions thrown by methods in that same controller.
  - Returns a `ResponseEntity` or a simple object that gets serialized to the response body.
  - **Example (within the same controller):**

```
@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity<String>
handleResourceNotFoundException(ResourceNotFoundException ex) {
    return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
}
```

- `@ControllerAdvice` & `@RestControllerAdvice`:
  - **Recommended:** These annotations centralize exception handling across multiple controllers. A class annotated with `@ControllerAdvice` (or `@RestControllerAdvice` for REST APIs) can contain `@ExceptionHandler` methods that apply globally to all controllers in the application.
  - **Benefits:** Avoids duplicating exception handling logic in every controller.
  - **Example:**

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;

@RestControllerAdvice // Applies to all @RestController classes
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse>
handleResourceNotFoundException(ResourceNotFoundException ex,
WebRequest request) {
        ErrorResponse error = new ErrorResponse(
            HttpStatus.NOT_FOUND.value(),
            ex.getMessage(),
            request.getDescription(false)
        );
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class) // Catch all other unhandled
exceptions
```

```
    public ResponseEntity<ErrorResponse>
handleGlobalException(Exception ex, WebRequest request) {
        ErrorResponse error = new ErrorResponse(
            HttpStatus.INTERNAL_SERVER_ERROR.value(),
            "An unexpected error occurred: " + ex.getMessage(),
            request.getDescription(false)
        );
        return new ResponseEntity<>(error,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

// Custom ErrorResponse DTO
public class ErrorResponse {
    private int status;
    private String message;
    private String details;
    // Constructor, Getters
}
```

Generating Standard/Consistent Responses

For REST APIs, it's good practice to provide consistent response formats, especially for errors.

- **DTOs (Data Transfer Objects):** Instead of exposing your JPA entities directly, create specific DTOs for request and response bodies. This decouples your API contract from your internal data model.
- **`ResponseEntity`:** As discussed, use `ResponseEntity` to explicitly set status codes and headers.
- **Custom Error Response Structure:** Define a standard JSON structure for error responses (e.g., including `timestamp`, `status`, `error message`, `path`). This makes error handling predictable for clients.

Example `ErrorResponse` DTO (from above):

```
public class ErrorResponse {
    private int status;
    private String message;
    private String details;
    private LocalDateTime timestamp;

    public ErrorResponse(int status, String message, String details) {
        this.status = status;
        this.message = message;
        this.details = details;
        this.timestamp = LocalDateTime.now();
    }
    // Getters for status, message, details, timestamp
}
```

Detailed Flow of Spring REST Service Execution

Understanding the request-response flow in a Spring REST service is key to troubleshooting and designing robust applications.

1. **Client Sends HTTP Request:** A client (e.g., web browser, mobile app, Postman) sends an HTTP request (GET, POST, PUT, DELETE) to a specific URL (e.g., `GET /api/products/1`).
2. **`DispatcherServlet`:**
    - The request first hits Spring's central servlet, the `DispatcherServlet`.
    - This is the "front controller" of the Spring Web MVC framework.
    - It's configured in `web.xml` (traditional Java EE) or automatically by Spring Boot.
3. **`HandlerMapping`:**
    - The `DispatcherServlet` consults a `HandlerMapping` (e.g., `RequestMappingHandlerMapping` for annotation-based controllers).
    - The `HandlerMapping` identifies the appropriate handler (controller method) based on the request URL, HTTP method, headers, etc.
4. **`HandlerAdapter`:**
    - Once the controller method is found, the `DispatcherServlet` dispatches the request to the `HandlerAdapter` (e.g., `RequestMappingHandlerAdapter`).
    - The `HandlerAdapter` is responsible for invoking the actual controller method.
    - It handles argument resolution (e.g., mapping `@PathVariable`, `@RequestParam`, `@RequestBody` to method parameters) and converts the return value.
5. **Controller Method Execution:**
    - The identified controller method (e.g., in a `@RestController`) is executed.
    - This method typically calls methods in the **Service Layer** to perform business logic.
6. **Service Layer Execution:**
    - The Service Layer contains the core business logic.
    - If the method is `@Transactional`, Spring's AOP proxy ensures it runs within a transaction.
    - The service method interacts with the **Repository Layer**.
7. **Repository Layer Execution:**
    - The Repository Layer (Spring Data JPA interface implementations) interacts with the persistence context (JPA `EntityManager`).
    - JPA/Hibernate translates method calls (e.g., `save()`, `findById()`, derived query methods) into **SQL queries**.
8. **Database Interaction:**
    - SQL queries are sent to the underlying **Database** via JDBC.
    - The database executes the queries and returns results.
9. **Data Flow Backwards:**
    - Database results are mapped back to JPA entities by Hibernate/JPA.
    - Entities are returned to the Repository Layer.
    - Entities are returned to the Service Layer.
    - Entities (or DTOs) are returned to the Controller Method.
10. **`HttpMessageConverter` (for `@RestController`):**
    - The `HandlerAdapter` (or Spring's default behavior for `@RestController`) uses an `HttpMessageConverter` (e.g., Jackson for JSON, JAXB for XML).
    - This converter serializes the Java object returned by the controller method into the appropriate format (e.g., JSON string).
11. **Response Generation:**

    - The `DispatcherServlet` creates the HTTP response, including the status code, headers, and the serialized response body.
12. **Client Receives Response:** The HTTP response is sent back to the client.

Content Negotiation

Content negotiation is the process by which a client and a server determine the best representation format (e.g., JSON, XML) for a given resource.

- **How Spring Handles It:** Spring Web MVC's `ContentNegotiationManager` facilitates this.
- **Mechanisms:**
    1. **`Accept` Header (HTTP Header, most common):** The client sends an `Accept` header in the request specifying the desired media types (e.g., `Accept: application/json`, `Accept: application/xml`). The server attempts to return data in the highest-priority acceptable format it supports.
    2. **Path Extension (Deprecated/Discouraged):** The client includes a file extension in the URL (e.g., `/products.json`, `/products.xml`). Spring maps this to the desired media type. (This method is generally discouraged due to security and URL cleanliness concerns).
    3. **Query Parameter:** The client includes a query parameter (e.g., `?format=json`). Spring can be configured to use this.
    4. **Default:** If no `Accept` header or other mechanism is provided, Spring typically defaults to JSON (with Jackson) in Spring Boot applications.

Configuration (e.g., in `application.properties` or `WebMvcConfigurer`):

```
# Disable path extension strategy (recommended)
spring.mvc.contentnegotiation.favor-path-extension=false
# Enable query parameter strategy (optional)
spring.mvc.contentnegotiation.favor-parameter=true
spring.mvc.contentnegotiation.parameter-name=format
```

Versioning RESTful Services

Versioning is essential for evolving APIs without breaking existing client applications. Strategies include:

1. **URI Versioning (Path Versioning):**
    - **Description:** Includes the version number directly in the URI path.
    - **Example:** `/api/v1/products`, `/api/v2/products`
    - **Pros:** Simple, clear, easy to cache, widely adopted.
    - **Cons:** Violates REST's principle of resources having a single URI, leads to URL proliferation.
    - **Implementation (Spring):** `@RequestMapping("/api/v1/products")` or `@GetMapping("/v1/products")`.
2. **Header Versioning (Custom Header):**
    - **Description:** Clients send a custom HTTP header indicating the desired API version.
    - **Example:** `X-API-VERSION: 1`
    - **Pros:** Resource URI remains clean, allows different versions of a resource under the same URI.

- **Cons:** Requires clients to understand and send custom headers, harder for browsers/caching proxies.
- **Implementation (Spring):** `@GetMapping(value = "/products", headers = "X-API-VERSION=1")`

3. **Accept Header Versioning (Media Type Versioning):**
   - **Description:** Uses the `Accept` HTTP header with a custom media type that includes the version.
   - **Example:** `Accept: application/vnd.company.app-v1+json`
   - **Pros:** Aligns with HTTP's content negotiation, single URI per resource.
   - **Cons:** More complex media types, clients need to construct specific `Accept` headers.
   - **Implementation (Spring):** `@GetMapping(value = "/products", produces = "application/vnd.company.app-v1+json")`
4. **Query Parameter Versioning:**
   - **Description:** Includes the version number as a query parameter in the URI.
   - **Example:** `/api/products?version=1`
   - **Pros:** Easy to use for clients, simple to implement.
   - **Cons:** Can be seen as polluting the URI, query parameters are for filtering/pagination, not resource identification.
   - **Implementation (Spring):** `@GetMapping(value = "/products", params = "version=1")`

Recommendation: URI versioning is often the easiest to implement and understand for most teams, despite violating a strict REST principle. Header or Accept Header versioning are more "RESTful" but can add client-side complexity.

Consuming REST Services in React Frontend

React applications (or any frontend) consume REST services using JavaScript's built-in `fetch` API or third-party libraries like `axios`.

- **`fetch` API:**
  - Native browser API (and available in React Native).
  - Returns a `Promise`.
  - Requires explicit JSON parsing (`response.json()`).
  - Does not automatically throw an error for HTTP error status codes (4xx/5xx).
- **`axios` Library:**
  - A popular, Promise-based HTTP client.
  - Automatically transforms JSON data.
  - Automatically handles HTTP error status codes (rejects Promise).
  - Provides interceptors for requests/responses (e.g., adding authentication tokens).

Example (using `fetch` in React):

```
import React, { useState, useEffect } from "react";

function ProductList() {
  const [products, setProducts] = useState([]);
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(true);
```

```
  useEffect(() => {
    fetch("http://localhost:8080/api/products") // Your Spring Boot API URL
      .then((response) => {
        if (!response.ok) {
          throw new Error(`HTTP error! status: ${response.status}`);
        }
        return response.json();
      })
      .then((data) => {
        setProducts(data);
        setLoading(false);
      })
      .catch((error) => {
        setError(error);
        setLoading(false);
      });
  }, []); // Empty dependency array means run once on mount

  if (loading) return <div>Loading products...</div>;
  if (error) return <div>Error: {error.message}</div>;

  return (
    <div>
      <h1>Products</h1>
      <ul>
        {products.map((product) => (
          <li key={product.id}>
            {product.name} - ${product.price}
          </li>
        ))}
      </ul>
    </div>
  );
}

export default ProductList;
```

- **CORS (Cross-Origin Resource Sharing):** When your React frontend (e.g., running on `localhost:3000`) tries to access a Spring Boot backend (e.g., running on `localhost:8080`), it's a cross-origin request. Browsers enforce the Same-Origin Policy, which prevents such requests by default for security.
    - **Solution:** Your Spring Boot backend must explicitly allow requests from your React frontend's origin using CORS headers.
    - **Spring Boot CORS:**
        - **Global Configuration:**

```
import org.springframework.context.annotation.Configuration;
import
```

```
org.springframework.web.servlet.config.annotation.CorsRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer
;

@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**") // Apply CORS to all /api
paths
                .allowedOrigins("http://localhost:3000",
"https://your-frontend-domain.com") // Allowed origins
                .allowedMethods("GET", "POST", "PUT", "DELETE",
"PATCH", "OPTIONS") // Allowed HTTP methods
                .allowedHeaders("*") // Allowed headers
                .allowCredentials(true); // Allow sending
cookies/auth headers
    }
}
```

- **Per-Controller/Method:** `@CrossOrigin(origins = "http://localhost:3000")`

Unit Testing of Repository Layer

Unit testing the Repository layer typically involves testing its interaction with the database. Spring Boot provides `@DataJpaTest` for this.

- `@DataJpaTest`**:**
  - **Purpose:** A specialized test annotation that focuses solely on JPA components. It auto-configures an in-memory database (like H2) and a `TestEntityManager` for testing. It also scans for `@Repository` beans.
  - **Scope:** Only loads components relevant to JPA tests, making tests fast.
  - `TestEntityManager`**:** Allows you to directly interact with the JPA `EntityManager` to persist entities before testing your repository methods.
- **Mocking:** Not typically used directly for `JpaRepository` methods as they are handled by Spring Data. You might mock dependent services if a repository had complex custom logic.

Example:

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;
import static org.assertj.core.api.Assertions.assertThat;
import java.util.List;
import java.util.Optional;

@DataJpaTest // Configures H2, scans @Repository beans
public class ProductRepositoryTest {
```

```
    @Autowired
    private ProductRepository productRepository; // Your repository interface

    @Autowired
    private TestEntityManager entityManager; // For setup/teardown

    @Test
    void whenFindByName_thenReturnProduct() {
        // Given
        Product apple = new Product("Apple", 1.0, 100);
        entityManager.persistAndFlush(apple); // Persist using TestEntityManager

        // When
        Optional<Product> found = productRepository.findByName("Apple");

        // Then
        assertThat(found).isPresent();
        assertThat(found.get().getName()).isEqualTo(apple.getName());
    }

    @Test
    void whenFindByPriceGreaterThan_thenReturnProducts() {
        // Given
        Product p1 = new Product("A", 10.0, 10);
        Product p2 = new Product("B", 20.0, 20);
        Product p3 = new Product("C", 5.0, 5);
        entityManager.persist(p1);
        entityManager.persist(p2);
        entityManager.persist(p3);
        entityManager.flush();

        // When
        List<Product> products = productRepository.findByPriceGreaterThan(15.0);

        // Then
        assertThat(products).hasSize(1);
        assertThat(products.get(0).getName()).isEqualTo("B");
    }
}
```

Unit Testing of Controller Layer Mocking Service Layer

Unit testing controllers involves testing their logic in isolation from the service layer and database. This is achieved by mocking the service layer dependencies. Spring Boot provides `@WebMvcTest` for this.

- `@WebMvcTest`:
  - **Purpose:** Focuses on Spring MVC components. It auto-configures `MockMvc` and scans `@Controller`, `@RestController`, `@ControllerAdvice`, etc. It does *not* load the full application context or `@Service`/`@Repository` beans.
  - **Scope:** Ideal for testing web layer slices without involving persistence.

    - `MockMvc`: A powerful utility for performing HTTP requests against your controller and asserting responses without starting a real HTTP server.
- `@MockBean`:
    - **Purpose:** Replaces a real bean in the Spring `ApplicationContext` with a Mockito mock.
    - **Use Case:** Essential for isolating the controller by providing mock implementations of its service dependencies. You then define the behavior of these mocks using Mockito's `when().thenReturn()` or `doNothing().when()`.

Example:

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import com.fasterxml.jackson.databind.ObjectMapper;
import java.util.Arrays;
import java.util.Optional;

import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.when;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@WebMvcTest(ProductRestController.class) // Test ProductRestController only
public class ProductRestControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean // Mock the ProductService dependency
    private ProductService productService;

    @Autowired
    private ObjectMapper objectMapper; // For converting objects to JSON

    @Test
    void whenGetProducts_thenReturnProductList() throws Exception {
        Product p1 = new Product(1L, "Laptop", 1200.0, 10);
        Product p2 = new Product(2L, "Mouse", 25.0, 50);
        when(productService.findAll()).thenReturn(Arrays.asList(p1, p2));

        mockMvc.perform(get("/api/products")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$[0].name").value("Laptop"))
                .andExpect(jsonPath("$[1].price").value(25.0));
    }

    @Test
```

```
    void whenGetProductById_thenReturnProduct() throws Exception {
        Product p1 = new Product(1L, "Laptop", 1200.0, 10);
        when(productService.findById(1L)).thenReturn(Optional.of(p1));

        mockMvc.perform(get("/api/products/1")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.name").value("Laptop"));
    }

    @Test
    void whenCreateProduct_thenReturnNewProduct() throws Exception {
        Product newProduct = new Product("Keyboard", 75.0, 20);
        Product savedProduct = new Product(3L, "Keyboard", 75.0, 20);
        when(productService.save(any(Product.class))).thenReturn(savedProduct);

        mockMvc.perform(post("/api/products")
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(newProduct))) // Convert
Product to JSON string
                .andExpect(status().isCreated()) // Expect 201 Created
                .andExpect(jsonPath("$.id").exists())
                .andExpect(jsonPath("$.name").value("Keyboard"));
    }
}
```

Integration Testing

Integration tests verify the interaction between different layers or components of your application (e.g., controller to service, service to repository, repository to database). They involve a larger portion of the application context.

- `@SpringBootTest`:
  - **Purpose:** Loads the entire Spring Boot application context. It can start an embedded server or use a mock environment.
  - **Options:**
    - `@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)`: Uses a mock web environment and `MockMvc` (similar to `@WebMvcTest` but loads full context).
    - `@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)`: Starts the application on a random available port. You can then use `TestRestTemplate` or `WebTestClient` to make actual HTTP calls.
  - **Scope:** Comprehensive testing, but slower due to loading the full context.
- **Database:** For integration tests involving the database, you might use a real database, an in-memory database (like H2), or a test container.

Example (`@SpringBootTest` with `MockMvc`):

```java
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import com.fasterxml.jackson.databind.ObjectMapper;
import static org.hamcrest.Matchers.hasSize;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@SpringBootTest // Loads full Spring Boot app context
@AutoConfigureMockMvc // Configures MockMvc
public class ProductIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private ProductRepository productRepository; // Real repository for
integration test

    @Autowired
    private ObjectMapper objectMapper;

    // Reset database before each test to ensure test isolation
    @BeforeEach
    void setUp() {
        productRepository.deleteAll();
        productRepository.save(new Product("Integration Laptop", 1500.0, 5));
        productRepository.save(new Product("Integration Mouse", 30.0, 20));
    }

    @Test
    void whenGetAllProducts_thenReturnsProducts() throws Exception {
        mockMvc.perform(get("/api/products")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$", hasSize(2))) // Expect 2 products from
setup
                .andExpect(jsonPath("$[0].name").value("Integration Laptop"));
    }

    @Test
    void whenCreateProduct_thenProductIsSaved() throws Exception {
        Product newProduct = new Product("Integration Keyboard", 100.0, 15);
        mockMvc.perform(post("/api/products")
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(newProduct)))
                .andExpect(status().isCreated())
                .andExpect(jsonPath("$.name").value("Integration Keyboard"));
```

```
        // Verify it's actually in the database
        assertThat(productRepository.findByName("Integration
Keyboard")).isPresent();
    }
}
```

Capstone Project Implementation

A capstone project is the culmination of your learning, where you apply all acquired knowledge to build a complete, functional application.

Typical Project Structure using Spring Boot, Spring Data JPA, and REST:

- **Build Tool:** Maven or Gradle
- **Database:** H2 (for development/testing), PostgreSQL/MySQL (for production deployment)
- **Layers:**
  - `src/main/java`:
    - `Application.java`: Main Spring Boot entry point (`@SpringBootApplication`).
    - `model` / `entity` **package:** JPA Entities (`@Entity`, `@Table`, `@Id`, `@GeneratedValue`, relationship annotations).
    - `repository` / `dao` **package:** Spring Data JPA Repository interfaces (`JpaRepository`, custom query methods).
    - `service` **package:** Service classes (`@Service`, `@Transactional`, business logic, orchestrating repository calls).
    - `controller` / `api` **package:** REST Controllers (`@RestController`, `@RequestMapping`, `@GetMapping`, `@PostMapping`, etc., handling HTTP requests, calling service layer).
    - `dto` **package:** Data Transfer Objects for request/response payloads (decoupling API from entities).
    - `exception` **package:** Custom exceptions.
    - `advice` **package:** Global exception handler (`@RestControllerAdvice`).
- `src/main/resources`:
  - `application.properties` / `application.yml`: Spring Boot configuration (database connection, server port, logging, JPA properties).
- `src/test/java`:
  - **Unit Tests:**
    - `repository` tests (`@DataJpaTest`).
    - `service` tests (often plain JUnit/Mockito without Spring context).
    - `controller` tests (`@WebMvcTest`, mocking service).
  - **Integration Tests:**
    - End-to-end tests for API endpoints (`@SpringBootTest`, `MockMvc` or `TestRestTemplate`).
- **Frontend (Optional but common):** A separate React (or Angular/Vue) project consuming the Spring Boot REST API.

Key Features to Implement in a Capstone Project:

- **CRUD Operations:** For at least two related entities (e.g., Products and Orders, Users and Posts).

- **Relationships:** Implement one-to-many, many-to-one, and potentially many-to-many relationships.
- **Pagination & Sorting:** Implement these on API endpoints.
- **Search/Filtering:** Implement custom query methods or JPQL queries.
- **Error Handling:** Consistent custom error responses for common scenarios (e.g., Not Found, Bad Request).
- **Validation:** Input validation using Jakarta Bean Validation (`@Valid`, `@NotNull`, `@Size`, etc.).
- **Transaction Management:** Ensure business operations are transactional.
- **Testing:** Comprehensive unit and integration tests.
- **Security (Basic):** (Optional but good to start considering) Spring Security for basic authentication/authorization.
- **Deployment:** Containerization (Docker) and deployment to a cloud platform (Heroku, AWS, Azure, GCP).

The capstone project brings all these pieces together into a working application, solidifying your understanding of how they integrate to form a complete solution.