|  |
| --- |
|  |
|  |
| Spring data jpa |
| **Response codes:** |
| 404 - not found |
| 400 - Bad request |
| 500 - Internal server error |
| 502 - Bad gateway |
| 403 - Forbidden |
| 408 - Request timeout |
| 302 - Found |
| 200 - OK |
| 405-not allowed |

**IOC :**

Inversion of Control (IoC) is a key principle in software engineering and is heavily used in the Spring Framework. In simple terms, IoC means that the control over object creation and management is inverted from the application to a framework or container.

In the context of Spring, IoC is achieved through dependency injection (DI). Dependency injection is a design pattern where the objects are provided with their dependencies instead of creating them internally.

Here's how IoC and dependency injection work in Spring:

1. **Dependency Injection**: Instead of manually creating objects or dependencies inside your classes, you declare the dependencies as attributes of the class. Spring's IoC container then injects these dependencies into the class at runtime. This way, the class doesn't need to worry about creating or managing its dependencies.
2. **Inversion of Control Container**: Spring's IoC container is responsible for managing the lifecycle of objects, creating and injecting dependencies, and wiring components together. When your application starts up, the Spring container reads the configuration metadata (typically in XML or annotations) and instantiates the objects defined in the configuration, injecting dependencies where required.
3. **Configuration Metadata**: In Spring, you can define the configuration metadata in XML files, Java annotations, or Java configuration classes. This metadata tells the Spring container which classes to instantiate, which dependencies to inject, and how to wire everything together.
4. **Bean Definition**: In Spring, a bean is an object that is managed by the Spring IoC container. The bean definition tells the container how to create and configure the bean. You can define beans in XML files using **<bean>** tags or in Java configuration classes using **@Bean** annotations.
5. **Component Scanning**: Spring also supports component scanning, where it automatically detects classes annotated with **@Component**, **@Service**, **@Repository**, etc., and registers them as beans in the IoC container. This eliminates the need for explicit bean definitions in XML or Java configuration.

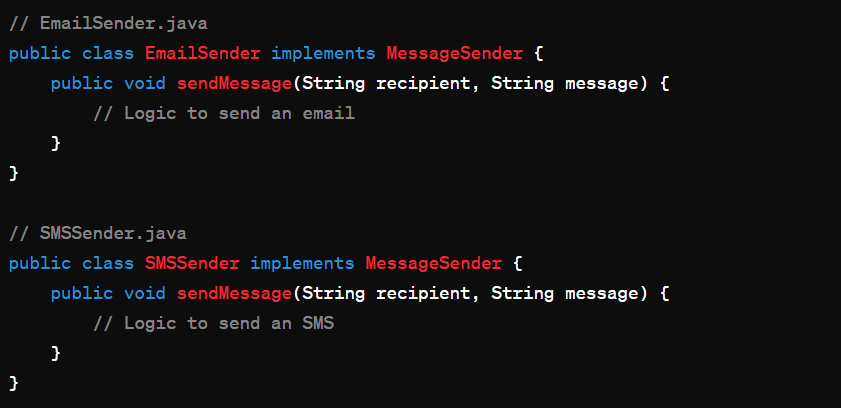
By following the principles of IoC and dependency injection, Spring promotes loose coupling between components, making your code more modular, easier to test, and maintainable. It also allows for better separation of concerns and promotes reusability of components.

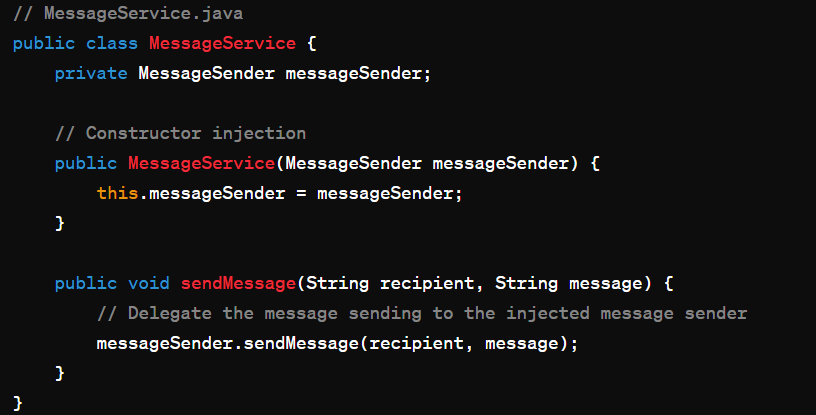
**Real time example :**

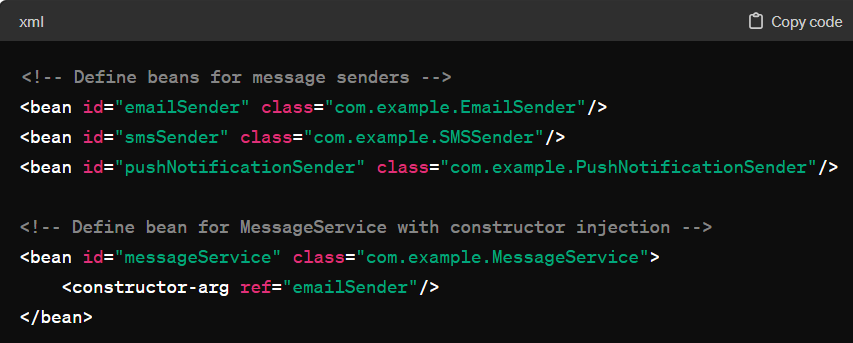
Let's consider a real-world example of a messaging application, where we have a **MessageService** that sends messages to different types of recipients such as email, SMS, and push notifications. We'll illustrate how Inversion of Control (IoC) and Dependency Injection (DI) can be applied in this scenario using Spring.

1. **Inversion of Control (IoC)**: In traditional programming, the **MessageService** class might have been responsible for creating instances of the various message sender classes (e.g., **EmailSender**, **SMSSender**, **PushNotificationSender**) itself. With IoC, the control over creating and managing these dependencies is inverted to the Spring framework.
2. **Dependency Injection (DI)**: The **MessageService** class declares dependencies on interfaces (e.g., **MessageSender**) rather than concrete implementations. Spring's IoC container then injects the appropriate implementations at runtime, based on the configuration.

Here's how it looks in code:

****

****

****

With this configuration, when Spring initializes the **MessageService** bean, it automatically injects the **EmailSender** instance into it. If we later want to switch to a different message sender (e.g., SMS or push notifications), we can simply update the configuration without modifying the **MessageService** class.

This separation of concerns, where the responsibility for creating and managing dependencies is handled by the Spring framework, illustrates the principles of IoC and DI in action.

**Loose Coupling vs Tight Coupling :**

Loose coupling and tight coupling are two different design concepts in software engineering, describing the degree of dependency between components or modules in a system.

1. **Loose Coupling**:
   * In a loosely coupled system, components or modules have minimal or no direct dependency on each other.
   * Changes in one module have little to no impact on other modules.
   * Communication between modules is typically done through interfaces or abstract contracts, allowing for flexibility and easy substitution of implementations.
   * Loose coupling promotes better modularity, reusability, and maintainability of the codebase.
   * Example: In an application with a service layer and a data access layer, the service layer may rely on interfaces provided by the data access layer rather than specific implementations, allowing different data access implementations (e.g., SQL, NoSQL) to be easily swapped without affecting the service layer.
2. **Tight Coupling**:
   * In a tightly coupled system, components or modules have strong dependencies on each other.
   * Changes in one module may require changes in multiple other modules.
   * Direct references between modules are common, leading to high interdependence.
   * Tight coupling can make the codebase more rigid and difficult to maintain or extend.
   * Example: A class that instantiates and directly uses objects of another class without abstraction or interfaces is tightly coupled to that class. Any changes in the implementation of the second class may require modifications in the first class.

In general, loose coupling is preferred in software design because it enhances flexibility, scalability, and maintainability. It allows for easier testing, promotes code reuse, and reduces the risk of unintended consequences when making changes to the system. Tight coupling, on the other hand, can lead to code that is harder to understand, modify, and maintain, making it more error-prone in the long run.

**MVC :**

MVC stands for Model-View-Controller, which is a software architectural pattern commonly used in web development. It separates an application into three interconnected components: the Model, the View, and the Controller. Here's a brief explanation of each component:

1. **Model**: Represents the data and business logic of the application. It encapsulates the application's state and behavior. The Model interacts with the database or any other data source to fetch or update data.
2. **View**: Represents the user interface of the application. It is responsible for displaying data to the user and capturing user input. Views are typically composed of HTML, CSS, and client-side scripting languages such as JavaScript.
3. **Controller**: Acts as an intermediary between the Model and the View. It handles user input and updates the Model accordingly. It also updates the View based on changes in the Model. Controllers contain application logic, routing requests to the appropriate Model methods and rendering the appropriate View.

Here's a real-time example to illustrate MVC:

Consider a simple web application for managing tasks. The application allows users to create, update, and delete tasks.

* **Model**: The Model represents the tasks in the application. It defines a Task class with attributes such as task name, description, due date, etc. It also provides methods to interact with the database, such as fetching tasks, adding new tasks, updating tasks, and deleting tasks.
* **View**: The View represents the user interface of the application. It consists of HTML templates that display the tasks to the user. For example, there could be a "tasks.html" template that lists all tasks along with options to add, edit, or delete tasks.
* **Controller**: The Controller handles user requests and updates the Model accordingly. It contains methods to handle actions such as displaying the list of tasks, adding a new task, editing an existing task, and deleting a task. For example, when a user submits a form to add a new task, the Controller receives the request, creates a new Task object, and saves it to the database through the Model. Then, it redirects the user to the updated list of tasks.

In this example, the Model represents the tasks and their management logic, the View represents the HTML templates for displaying tasks, and the Controller handles user requests and updates the Model and View accordingly. The separation of concerns provided by the MVC pattern makes the application easier to maintain, extend, and test.

**Thymeleaf :**

Thymeleaf is a modern server-side Java template engine for web and standalone environments. It allows developers to create dynamic web pages with natural, human-readable templates. Thymeleaf seamlessly integrates with Spring Framework and other Java web frameworks.

**Internationalization :**

Internationalization, often abbreviated as i18n (because there are 18 letters between the "i" and the "n" in "Internationalization"), is the process of designing and developing software applications in a way that makes them adaptable to different languages, regions, and cultural conventions without engineering changes.

In simpler terms, internationalization means making a software application capable of being used by people from different parts of the world who speak different languages or have different cultural norms. This involves separating the text and other elements that might need to change based on language or location from the core code of the application.

For example, in an internationalized application, the text displayed to the user would be stored separately from the application code, allowing it to be easily translated into different languages. Additionally, the application might adapt its behavior based on the user's location or cultural settings, such as displaying dates and currency in the appropriate format for that region.

By internationalizing a software application, developers can reach a wider audience and provide a better user experience for people around the world, regardless of their language or cultural background.

**Hibernate :**

Hibernate is a Java-based framework that simplifies the interaction between Java applications and relational databases. It provides an object-relational mapping (ORM) approach, allowing developers to work with objects in their Java code instead of dealing directly with database tables and SQL queries.

Here's a simple explanation of how Hibernate works:

1. **Object-Relational Mapping (ORM)**: Hibernate maps Java classes to database tables and Java data types to SQL data types. This mapping is defined using annotations or XML configuration files.
2. **Entity Classes**: In Hibernate, entity classes represent the data stored in the database. Each entity class corresponds to a database table, and each instance of the entity class represents a row in that table.
3. **Session Factory**: Hibernate uses a Session Factory to create sessions, which are used to interact with the database. The Session Factory is typically configured with information about the database connection, mapping metadata, and other settings.
4. **Session**: A session in Hibernate represents a single unit of work with the database. Developers use sessions to perform database operations such as saving, updating, deleting, and querying objects.
5. **Transactions**: Hibernate supports transactions, which are sequences of database operations that are executed as a single unit. Transactions ensure data integrity by either committing all changes to the database or rolling back the changes if an error occurs.
6. **Lazy Loading**: Hibernate supports lazy loading, which means that it loads associated objects from the database only when they are accessed for the first time. This can improve performance by reducing the amount of data fetched from the database.

Overall, Hibernate simplifies database access in Java applications by abstracting away the complexities of JDBC (Java Database Connectivity) and providing a higher-level, object-oriented approach to interacting with relational databases. It automates common tasks such as mapping Java objects to database tables, managing database connections, and executing SQL queries, allowing developers to focus on business logic rather than low-level database operations.

**Spring REST :**

Spring REST, or Spring RESTful Web Services, is a part of the Spring Framework that facilitates the development of RESTful APIs in Java.

Here's a simple explanation:

1. **RESTful Architecture**: REST stands for Representational State Transfer. It's an architectural style for designing networked applications, especially web services, that prioritize simplicity, scalability, and reliability. RESTful APIs use standard HTTP methods like GET, POST, PUT, DELETE to perform CRUD (Create, Read, Update, Delete) operations on resources.
2. **Spring REST**: Spring REST allows developers to build RESTful APIs using the Spring Framework. It provides features and components to simplify the creation of RESTful web services, handling request mapping, serialization, deserialization, error handling, and more.
3. **Controller Classes**: In Spring REST, you create controller classes to define the endpoints of your RESTful API. These controllers handle incoming HTTP requests and return appropriate responses. You can use annotations like **@RestController**, **@RequestMapping**, **@GetMapping**, **@PostMapping**, etc., to define the URL mappings and HTTP methods.
4. **Data Transfer Objects (DTOs)**: Spring REST often uses Data Transfer Objects (DTOs) to represent the data exchanged between the client and the server. These DTOs are simple Java classes that define the structure of the data being sent or received.
5. **Serialization and Deserialization**: Spring REST automatically serializes Java objects to JSON or XML format when sending responses to clients, and deserializes JSON or XML payloads into Java objects when receiving requests from clients. This process is handled by libraries like Jackson or JAXB, which are integrated into Spring.
6. **Error Handling**: Spring REST provides mechanisms for handling errors and exceptions that occur during API requests. Developers can define exception handlers to customize error responses sent back to clients.

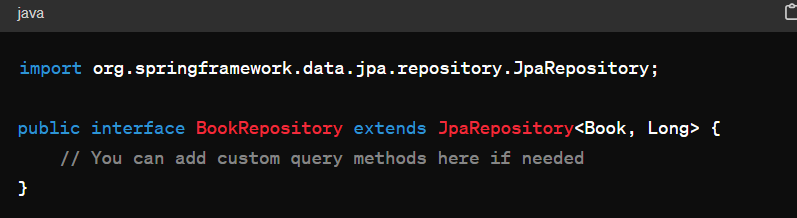
In summary, Spring REST simplifies the process of building RESTful APIs in Java by providing a set of features and components that handle common tasks such as request mapping, serialization, deserialization, and error handling. It allows developers to focus on defining the API endpoints and implementing business logic, while Spring takes care of the underlying infrastructure and boilerplate code.

**Spring Data JPA :**

Spring Data JPA is part of the larger Spring Data project, which aims to simplify data access in Java applications, particularly in Spring-based projects.

Here's a simple explanation of Spring Data JPA:

**JPA**: JPA stands for Java Persistence API, which is a Java specification for managing relational data in applications. It provides a set of standard APIs for object-relational mapping (ORM) in Java applications.

****

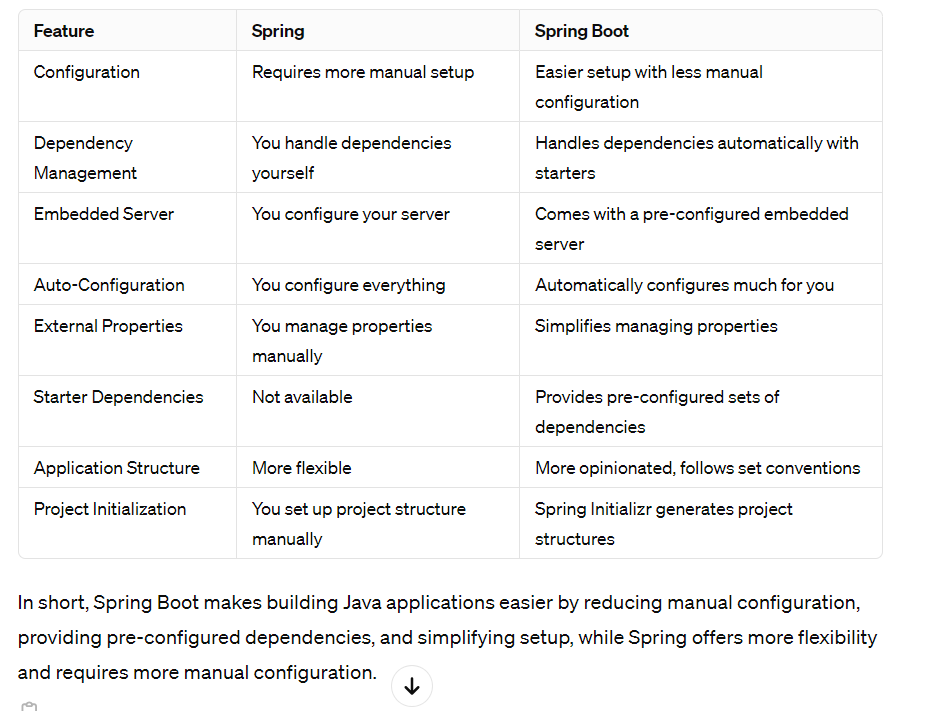
In this example, we have a Spring Boot application with a **Book** entity, a **BookRepository** interface extending **JpaRepository**, a **BookService** class to perform CRUD operations, and a **BookController** class to handle HTTP requests. Spring Data JPA simplifies data access by automatically generating queries based on method names and providing common CRUD operations out of the box.

Spring Data JPA provides a set of repository interfaces and methods that allow you to perform common data access operations without writing boilerplate code. By extending the JpaRepository interface, you gain access to a variety of predefined methods. Here are some of the common methods available:

**Common Methods from JpaRepository---**

1. **Saving and Updating Entities**
   * save(S entity): Saves a given entity.
   * saveAll(Iterable<S> entities): Saves all given entities.
2. **Deleting Entities**
   * deleteById(ID id): Deletes the entity with the given ID.
   * delete(T entity): Deletes a given entity.
   * deleteAll(Iterable<? extends T> entities): Deletes the given entities.
   * deleteAll(): Deletes all entities managed by the repository.
3. **Finding Entities**
   * findById(ID id): Retrieves an entity by its ID.
   * findAll(): Returns all instances of the type.
   * findAllById(Iterable<ID> ids): Returns all instances of the type with the given IDs.
   * findAll(Sort sort): Returns all entities sorted by the given options.
   * findAll(Pageable pageable): Returns a paginated list of entities.
4. **Counting Entities**
   * count(): Returns the number of entities available.
5. **Checking Existence**
   * existsById(ID id): Checks whether an entity with the given ID exists.

**Spring vs Spring boot :**

****