***Module 9) Java – Spring***

1. Introduction to Spring Framework

**Theory:**

**What is Spring Framework?**

* Overview of the Spring Framework and its purpose in Java development.
* The Spring Framework is an open-source framework for building Java applications. It helps developers create scalable, secure, and maintainable applications.

Purposes:

* **Simplified Development**: Spring reduces boilerplate code and offers abstractions for common tasks such as database access, transaction management, and web development.
* **Loose Coupling**: Through dependency injection, Spring enables loosely coupled components, making applications easier to test and maintain.
* **Testability**: With its DI and modular architecture, Spring makes it easier to perform unit and integration tests.
* **Flexibility and Integration**: Spring integrates well with a variety of databases, messaging systems, and web technologies, making it highly adaptable to different environments.
* **Modular and Scalable**: Spring's modular architecture means developers can pick and choose only the features they need, making it highly scalable for both small and large applications.

**Key features of Spring:**

* Inversion of Control (IoC): The **Inversion of Control (IoC)** is a core concept of the **Spring Framework**, and it primarily refers to the way in which the control of object creation and dependency management is inverted. In traditional programming, objects are responsible for creating and managing their dependencies. With IoC, the framework takes control of these responsibilities, allowing objects to be loosely coupled and more flexible.
* Dependency Injection (DI): IoC in Spring is primarily implemented through **Dependency Injection** . DI allows Spring to manage and inject the necessary dependencies into objects (beans) at runtime, rather than having the objects create or manage their dependencies themselves.
* Aspect-Oriented Programming (AOP): **Aspect-Oriented Programming (AOP)** is a powerful programming paradigm that complements Object-Oriented Programming (OOP) by allowing separation of concerns in a way that addresses cross-cutting concerns (aspects). These are concerns that affect multiple parts of an application, such as logging, security, transaction management, error handling, and caching .In the context of the **Spring Framework**, AOP is used to provide a clean, modular way of handling these cross-cutting concerns without cluttering the business logic.
* Transaction Management: transaction management in Spring is a key part of ensuring that database operations (or any other resource operations) are handled in a consistent and reliable manner. Spring provides support for both **programmatic** and **declarative** transaction management, abstracting away the underlying transaction APIs (e.g., JTA, JDBC, Hibernate, etc.) and providing a unified way to manage transactions.
* Spring's flexibility for creating both web and non-web applications: For web applications, Spring offers robust tools like Spring MVC, Spring WebFlux, and Spring Security. For non-web applications, it provides support for batch processing, microservices, scheduling, and more. Additionally, its core features like Dependency Injection, AOP, and transaction management are essential for building both types of applications, making Spring a versatile choice for a wide range of use cases.

**Spring Architecture**:

Overview of the core components of the Spring Framework:

* **Core Container**: IoC and DI

**Core Container: IoC and DI**

1. **Inversion of Control (IoC):**
   * **Definition:** IoC refers to the concept where the control of creating and managing objects is inverted from the traditional approach. Instead of the application code controlling how objects are created and managed, the Spring container takes over this responsibility.
   * **Purpose:** This helps to decouple the components of an application, making it easier to manage dependencies and enhancing modularity and flexibility.
   * **How It Works:** In Spring, the IoC container is responsible for instantiating, configuring, and assembling objects. These objects (beans) are defined in configuration files or annotations, and the container injects them into the appropriate places within the application.
2. **Dependency Injection (DI):**
   * **Definition:** DI is a design pattern used to implement IoC, where the dependencies (objects or services) required by a class are provided to it, rather than the class creating them itself.
   * **Purpose:** The main goal is to reduce tight coupling between components. By injecting dependencies from outside, classes don't need to manage their own dependencies, which leads to cleaner, more testable code.
   * **Types of Dependency Injection:**
     + **Constructor Injection:** Dependencies are provided through the constructor of the class.
     + **Setter Injection:** Dependencies are provided through setter methods.
     + **Field Injection:** Dependencies are injected directly into the fields of a class (typically through annotations).
   * **How It Works:** In Spring, the container manages the lifecycle of beans and injects dependencies as per the configuration (XML or annotations). The container can inject beans into classes based on the declared type, making it easier to manage and replace dependencies.

* **Spring AOP**: Aspect-Oriented Programming

**Aspect-Oriented Programming (AOP)** in the context of **Spring Framework** refers to a programming paradigm that allows you to separate cross-cutting concerns (concerns that affect multiple classes or modules) from the core business logic. Spring AOP provides a way to define aspects (modular pieces of code) that can be applied to various parts of an application without modifying the actual business logic.

* **Spring ORM**: Integrating Spring with ORM frameworks (e.g., Hibernate, JPA)

**Spring ORM** refers to Spring's support for integrating Object-Relational Mapping (ORM) frameworks like **Hibernate**, **JPA (Java Persistence API)**, **JDO (Java Data Objects)**, and others, with the Spring Framework. ORM frameworks are commonly used to manage database interactions and provide an object-oriented abstraction over relational databases.

**Hibernate** and **JPA** (Java Persistence API). Both are supported by Spring ORM, but **JPA** is the standard, and **Hibernate** is an implementation of JPA.

* **Spring Web**: Web framework for creating Java web applications.

**Spring Web** is a part of the **Spring Framework** that provides comprehensive support for building **web applications** using **Java**. It enables the development of both **traditional web applications** (using JSPs, servlets) and **modern RESTful APIs** (using REST, JSON, and other technologies). The Spring Web module offers a wide range of features, from handling HTTP requests and managing sessions to simplifying the development of web controllers and facilitating integration with various web technologies.

* **Spring MVC**: Model-View-Controller framework for building web applications.

**Spring MVC (Model-View-Controller)** is a comprehensive framework for building **web applications** within the **Spring Framework**. It follows the **MVC** design pattern to separate concerns in a web application, providing a clear distinction between the **model**, **view**, and **controller** components. This separation helps in organizing the code, making it more modular, testable, and maintainable.

*2. BeanFactory and ApplicationContext*

**Theory:**

* **BeanFactory vs. ApplicationContext**:
* What is **BeanFactory**?: the BeanFactory is a container that manages beans, their dependencies, and their lifecycle. It's a core component of Spring's Dependency Injection (DI) and Inversion of Control (IoC) features.
* Pros and cons of using BeanFactory:

1. **Lightweight and Efficient (Minimal Overhead)**:
   * **Low Memory Consumption**: BeanFactory is very lightweight and has a minimal memory footprint. It does not pre-initialize beans at startup; instead, it uses **lazy initialization**, which means beans are created only when they're actually needed. This can be advantageous in memory-constrained environments or when the application only needs a subset of beans during runtime.
   * **Ideal for Small, Lightweight Applications**: If your application is very simple or resource-constrained (e.g., an embedded system), BeanFactory can be more efficient compared to ApplicationContext since it doesn't load all the beans upfront.
2. **Lazy Initialization**:
   * By default, BeanFactory lazily initializes beans, meaning that beans are not created until they are explicitly requested. This can be helpful when you want to avoid unnecessary bean creation for parts of the application that may not be used at all.
   * Lazy initialization can save resources (e.g., CPU, memory) in certain use cases, especially when only a limited subset of beans is required.
3. **Decoupling and Dependency Injection**:
   * Like ApplicationContext, BeanFactory also provides dependency injection (DI), helping decouple components and manage dependencies. You still get the benefit of Spring’s DI container, making your code easier to test, maintain, and extend.
4. **Access to Spring Features (with some limitations)**:
   * BeanFactory still offers many of the basic Spring features such as bean lifecycle management, scope management, and basic dependency injection. You can use BeanFactory in scenarios where you need a very basic DI container without the full overhead of ApplicationContext.

**Cons of Using BeanFactory:**

1. **Limited Features Compared to ApplicationContext**:
   * **Missing Event Handling**: One of the biggest disadvantages of BeanFactory is that it doesn’t support event propagation, meaning it cannot listen to application events (such as custom events or lifecycle events).
   * **No Internationalization Support**: Unlike ApplicationContext, BeanFactory does not provide built-in support for internationalization (i18n) and message resolution.
   * **No AOP Support**: BeanFactory lacks support for **Aspect-Oriented Programming (AOP)**, which is available in ApplicationContext. AOP is commonly used in Spring for aspects like transactions, logging, and security.
   * **No Auto-wiring or Annotation Support (in early versions)**: In older versions of Spring, BeanFactory did not support features like annotation-based dependency injection (@Autowired, @Component, etc.), which are available in ApplicationContext. While newer versions of BeanFactory may support this, it is still far less powerful and flexible compared to ApplicationContext.
2. **No Support for ApplicationContext Features**:
   * **Environment and Property Resolution**: ApplicationContext provides rich support for externalized configuration, environment abstraction, and resolving properties, which is not available in BeanFactory.
   * **No Bean Post Processors**: ApplicationContext can register **BeanPostProcessors**, which allow you to modify beans before or after initialization. BeanFactory lacks this functionality.
   * **No BeanFactoryPostProcessor**: ApplicationContext supports BeanFactoryPostProcessor, which allows you to modify bean definitions before the container is initialized. BeanFactory does not provide this functionality.
3. **No Easy Access to Spring's Built-in Services**:
   * With ApplicationContext, you can easily access features such as **MessageSource** (for i18n support), **ApplicationEventPublisher**, and **ResourceLoader**. These services are either unavailable or much harder to access through BeanFactory.
4. **Less Convenient than ApplicationContext**:
   * **No Bean Lookup by Type**: ApplicationContext allows you to retrieve beans by type, name, or with annotations like @Autowired or @Qualifier. BeanFactory is generally less flexible and user-friendly compared to ApplicationContext in this regard.
   * **No Support for Auto-wiring**: While BeanFactory can inject dependencies, it lacks the more advanced and user-friendly auto-wiring capabilities (@Autowired, @Inject) found in ApplicationContext.
5. **Deprecated in Some Scenarios**:
   * BeanFactory is considered the **basic container** for Spring, while **ApplicationContext** is the **more feature-rich** and preferred container for most Spring-based applications. As a result, many of Spring’s newer features (like transaction management, AOP, and event handling) are only available in ApplicationContext.
   * **Spring Boot**, which simplifies Spring application configuration and setup, doesn't typically use BeanFactory directly; it uses ApplicationContext in most scenarios.

* What is **ApplicationContext**?: A more advanced container that includes features like event propagation,declarative mechanisms, and AOP support.
* Differences between **BeanFactory** and **ApplicationContext** (e.g., lazy initialization in

BeanFactory vs. eager initialization in ApplicationContext).

 **BeanFactory**:

* **Lazy Initialization** by default.
* Beans in BeanFactory are created only when they are requested. This means the beans are not created at startup; they are instantiated only when they are explicitly accessed or needed.
* This is beneficial in scenarios where you want to delay the creation of beans to save resources, especially in large applications where not all beans are required at startup.

 **ApplicationContext**:

* **Eager Initialization** by default.
* Beans in ApplicationContext are created at the time of container startup (unless specifically configured for lazy initialization). This ensures that all the beans are ready for use immediately after the context is loaded, which can be useful for applications that need to perform initialization tasks as soon as they start.
* In an ApplicationContext, the beans are instantiated early, typically allowing for a faster response when they are needed at runtime.

**Spring Beans**:

* Definition of a bean in Spring: In **Spring Framework**, a **bean** is essentially an **object** that is managed by the Spring IoC (Inversion of Control) container. The term "bean" is used to refer to any object that Spring instantiates, configures, and manages throughout its lifecycle. Beans are the fundamental building blocks of a Spring application.
* Scope of beans: Singleton, Prototype, Request, Session:

In **Spring**, **bean scopes** determine the lifespan and visibility of a bean within the Spring context. Each scope defines how many instances of a bean exist and how they are shared or created based on specific conditions. Let's look at the most common scopes:

**1. Singleton Scope (Default Scope)**

* + **Singleton** is the **default scope** in Spring. Only **one instance** of the bean is created for the entire Spring IoC container (application context). This instance is shared across all requests and usages within the application.
  + The singleton bean is created when the **ApplicationContext** is initialized, and it is destroyed when the context is closed (for example, on application shutdown).

**2. Prototype Scope**

* **Description**:
  + When a bean is configured with the **prototype** scope, Spring will create a **new instance** of the bean every time it is requested from the container.
  + The Spring container does **not** manage the lifecycle of prototype-scoped beans (e.g., it won't call the destroy method when the application shuts down). The bean is created whenever requested, and it is the responsibility of the caller to manage the bean's lifecycle (including destruction, if necessary).

**3. Request Scope**

* **Description**:
  + **Request scope** is used primarily in **web applications**. A new instance of the bean is created for each **HTTP request** and it is destroyed once the request has been processed.
  + The bean lives only for the duration of a single HTTP request and is shared across the components involved in processing that request. Each new HTTP request gets a fresh instance of the bean.

**4. Session Scope**

* **Description**:
  + **Session scope** is also used in **web applications**. A new instance of the bean is created for each **HTTP session**. The bean lives for the duration of a user's session, typically as long as the user remains active or until the session expires.
  + The bean is scoped to a user's session and is shared across the components that are processing that session. It is **not** shared across sessions.

*3. Container Concepts in Spring:*

**Theory:**

* **Spring IoC (Inversion of Control)**:
* Understanding IoC and how Spring uses it to manage object creation and dependencies: **nversion of Control (IoC)** is a design principle in software engineering in which the control over the flow of execution is inverted or reversed. Traditionally, in a typical program, the main method or control flow decides when and how objects are created. However, with IoC, the control is transferred to an external framework or container (such as Spring), which manages the object creation and their lifecycle.In simpler terms, **IoC means that the framework (Spring) is responsible for managing the creation and management of objects, instead of the application directly creating and managing them.**
* **Object Creation**: Instead of manually creating objects using new (as is common in traditional programming), Spring’s IoC container instantiates objects for you based on configuration (XML, annotations, or Java config).
* **Dependency Injection (DI)**: Spring injects dependencies (other objects) into beans rather than having the beans create or look for their dependencies. This is typically done via constructor injection, setter injection, or field injection. DI is one of the core principles of IoC.
* Benefits of IoC in application design (loose coupling, modularity, and testability).
* Benefits of Loose Coupling:
* **Flexibility**: You can easily change implementations of dependencies without affecting other parts of the application. For example, switching from a SQL database to a No SQL database only requires changing configurations in the Spring container rather than updating each dependent class.
* **Easier Maintenance**: Since components are less interdependent, it becomes easier to make changes in one part of the application without risking side effects on other parts.
* **Separation of Concerns**: Each component is responsible for a specific piece of functionality, without being concerned with how dependencies are created or managed.
* Benefits of Modularity:
* **Reusability**: Since components are modular, they can be reused in different parts of the application or even in different applications entirely.
* **Scalability**: Modular design makes it easier to scale the application by adding more independent components or changing existing ones.
* **Better Organization**: Modular design results in a clean and organized codebase, where each class or module has a single responsibility and can be worked on independently.
* Benefits of Testability:
* **Faster Development**: With IoC, developers can write unit tests early in the development cycle, ensuring that bugs are caught early and reducing the cost of fixing them.
* **High Test Coverage**: Since components are loosely coupled, it’s easier to achieve high test coverage because individual components can be tested in isolation.
* **Confidence in Refactoring**: With comprehensive unit tests in place, developers can refactor the code without fear of breaking existing functionality, knowing that the tests will catch any regressions.

**Dependency Injection (DI)**:

* Types of Dependency Injection:
* Constructor-based Dependency Injection: **Constructor-based Dependency Injection (Constructor Injection)** is one of the most commonly used forms of dependency injection in Spring. In this approach, **dependencies are provided to a class via its constructor**. This allows the class to receive the required dependencies when it is instantiated by the Spring container, ensuring that the class is always created with all necessary dependencies.

In constructor-based DI, the **Spring container** injects the dependencies into the class at the time of bean creation, making the dependencies **immutable** (i.e., they can't be changed after the object is created). This is useful for ensuring that an object is always in a valid state, as all necessary dependencies must be provided upfront.

* Setter-based Dependency Injection: **Setter-based Dependency Injection (Setter Injection)** is another form of dependency injection where the Spring container **injects dependencies through setter methods** after the bean is instantiated. Unlike **constructor injection**, where dependencies are passed via the constructor during object creation, setter injection allows dependencies to be set via **setter methods** after the object is created.

In this approach, the dependencies are **optional** (can be set or changed after the object is created), and the class can still be instantiated even if some dependencies are not provided. This makes setter injection more flexible compared to constructor-based injection, but it can also lead to objects being in an inconsistent state if dependencies are not correctly set.

* Advantages of DI in Spring.

### **Loose Coupling:** DI helps achieve **loose coupling** between components. When dependencies are injected by an external container (like Spring), rather than being directly created within the class, components do not need to know about each other’s internal workings.

### **Modularity and Reusability:** DI promotes ****modular design**** by encouraging the development of independent components that can be easily replaced or extended.

### **Easier Unit Testing:** DI significantly enhances the ****testability**** of the application. Since dependencies are injected into the class rather than being tightly coupled, it’s easy to swap real implementations with mock objects or stubs during unit testing.

### **Flexibility and Configurability :**DI makes it very easy to ****configure and manage**** your application components. By using ****Spring's IoC container****, you can configure different behaviors for beans, choose implementations dynamically, and switch dependencies without altering the actual code.

### **Centralized Configuration :**With DI in Spring, you can ****centralize your application configuration**** into one location, making it easier to manage and modify.

### **Improved Code Maintainability:**DI supports the ****separation of concerns**** principle, which makes your code easier to maintain.

### **Reduced Boilerplate Code:**In traditional object-oriented programming, managing dependencies requires manually creating and passing objects. DI helps to ****reduce boilerplate code**** by letting the container handle dependency management for you.

### *4. Spring Data JPA Template*

**Theory:**

**What is Spring Data JPA?**

* Introduction to Spring Data JPA and how it simplifies interaction with databases.
* **Spring Data JPA** is part of the **Spring Data** family and is a powerful framework that simplifies database interaction in Java applications. It provides a high-level abstraction for **Java Persistence API (JPA)**, which is a specification for object-relational mapping (ORM) in Java. Spring Data JPA enables developers to work with **relational databases** using **JPA entities** without needing to manually write boilerplate database code.
* Spring Data JPA builds on top of JPA and offers a **repository-based approach** to data access. It significantly reduces the need for boilerplate code when working with JPA, enabling developers to focus on defining the **data model** and **business logic** rather than managing CRUD operations or query logic.

### ****How Spring Data JPA Simplifies Database Interaction****

1. **Reduces Boilerplate Code**: By providing built-in implementations of common operations like save(), findById(), and delete(), Spring Data JPA reduces the need to write boilerplate code for interacting with the database.
2. **Automatic Query Generation**: The framework can automatically generate queries based on the names of the methods, meaning you don't have to write SQL or JPQL for simple queries.
3. **Declarative Querying**: Custom queries are simple with @Query and can be declared declaratively using annotations. No need for manually configuring EntityManager or writing custom SQL.
4. **Built-in Pagination and Sorting**: Spring Data JPA handles **pagination** and **sorting** with minimal configuration, making it easy to deal with large datasets without writing complex query logic.
5. **Extends JPA**: Since Spring Data JPA builds on top of JPA, you get all the benefits of JPA’s powerful ORM features, like automatic entity management, lazy loading, and dirty checking, but with less complexity.
6. **Integration with Spring Boot**: Spring Data JPA integrates seamlessly with **Spring Boot**, allowing for zero-configuration setups for repositories and making it easy to set up a data access layer in Spring-based applications.

* Explanation of JPA (Java Persistence API) and its role in ORM (Object Relational Mapping).
* **Java Persistence API (JPA)** is a **Java specification** that provides a standard for managing **relational data** in Java applications. It simplifies the interaction between Java objects (also known as **POJOs** or **Plain Old Java Objects**) and a relational database. JPA is a key technology for **Object-Relational Mapping (ORM)**, which is the technique used to map Java objects to database tables.
* JPA defines how **Java objects** (entities) can be **persisted** (stored) in a relational database, retrieved, and modified. It abstracts much of the boilerplate code needed for database interaction and integrates with other frameworks like **Hibernate**, **EclipseLink**, and **OpenJPA** to provide ORM functionality.
* **Object-Relational Mapping (ORM):**JPA plays a central role in ORM by mapping **Java objects** to **relational database tables** and facilitating the **automatic conversion** between Java types (objects) and database types (rows and columns).JPA abstracts the complexities of **SQL operations** and **database interactions** while providing a high-level API to work with data
* Benefits of using Spring Data JPA over manual SQL queries.

### ****Reduced Boilerplate Code****

### **Spring Data JPA** significantly reduces the amount of boilerplate code required for data access. You no longer need to write CRUD operations or manually create SQL queries for common operations like save(), find(), update(), and delete(). These methods are automatically generated by Spring Data JPA when you extend JpaRepository or CrudRepository.

### ****Abstraction of SQL Syntax****

### With **Spring Data JPA**, you don’t need to worry about writing SQL directly. Instead, you interact with **entities** and **repositories** in a high-level way, and Spring Data JPA translates your requests into SQL behind the scenes. This abstraction shields you from the intricacies of SQL and database-specific syntax.

### Less Error-Prone and Safer Code

### Manually writing SQL can often lead to errors, especially when dealing with complex queries, joins, or large databases. Spring Data JPA reduces the risk of mistakes in query construction because:

It **automates query creation** based on method names.

It uses **parameterized queries**, reducing the risk of **SQL injection attacks**.

### ****Better Integration with Spring Framework****

### **Spring Data JPA** integrates seamlessly with other Spring features, like **Spring Boot**, **Spring Transaction Management**, **Spring Security**, and **Spring’s Dependency Injection**. This integration makes it easier to manage database connections, transactions, and other concerns within your Spring-based application.

**Spring Data JPA Components**:

* **Repositories**: How Spring Data JPA auto-generates repository implementations.

Spring Data JPA auto-generates repository implementations by using a combination of **proxy-based** and **reflection-based** mechanisms. When you define an interface that extends one of the predefined Spring Data repository interfaces, Spring Data JPA automatically creates a proxy implementation of that interface at runtime.

* **Entities**: Mapping Java objects to database tables using JPA annotations.

Java Persistence API (JPA) provides a way to map Java objects (entities) to database tables, allowing you to interact with the database using object-oriented paradigms instead of SQL. This mapping is done using annotations, which tell JPA how to map Java classes to database tables and their fields to columns.

The @Entity annotation is used to mark a class as an entity, which means it will be mapped to a table in the database.

* **Query Methods**: Creating custom queries using method naming conventions (e.g.,

findById, findByName).

* Spring Data JPA derives queries based on the method name by parsing it into parts and applying these rules:

The method name starts with a keyword such as find, read, count, delete, or exists.

The keyword is followed by By and then the criteria or attributes that you want to query.

* **Method name**: findByFirstName
* **Generated Query**: SELECT u FROM User u WHERE u.firstName = ?

*5. Spring MVC*

**Theory:**

**What is Spring MVC?**:

* Overview of the MVC (Model-View-Controller) design pattern.
* The **Model-View-Controller (MVC)** design pattern is a widely used architectural pattern in software engineering for developing user interfaces. It divides the application into three main components, each responsible for a specific aspect of the application. This separation of concerns helps in making the application more modular, maintainable, and testable.
* Here's an overview of the three core components of the **MVC pattern**:

**Model:** The **Model** represents the **data** or **business logic** of the application.It is responsible for:

* **Storing data**: The model holds the data of the application, such as user information, database records, etc.
* **Business logic**: The model encapsulates the core logic of the application, such as data processing, calculations, and state transitions.
* **Interacting with the database**: It is often responsible for retrieving, saving, and updating data in the database (often using an ORM like JPA in Java-based applications).
* **Notifies the View**: The model notifies the view when the data changes, so the view can update accordingly (typically using patterns like **Observer**).

**View**: The **View** is the **user interface (UI)** of the application. It is responsible for:

* **Displaying data**: The view presents the data from the model in a format that the user can understand, such as HTML in web applications, or GUI components in desktop applications.
* **User interaction**: It handles user input (e.g., clicks, typing) and forwards it to the controller for processing.
* **Updating the UI**: The view listens for changes in the model and updates the UI accordingly, reflecting any data changes.

**Controller:** The **Controller** acts as an intermediary between the **Model** and the **View**. It is responsible for:

* **Processing user input**: The controller receives user input (usually from the view), processes it (sometimes by modifying the model), and decides which view should be displayed next.
* **Updating the model**: It can modify the model based on the input from the user.
* **Selecting the view**: After processing the input and possibly changing the model, the controller chooses the appropriate view to display the updated model.
* Explanation of the Spring MVC framework and how it simplifies web development.
* **Spring MVC** (Model-View-Controller) is a part of the larger **Spring Framework**, which is widely used for building robust, scalable, and flexible web applications in Java. Spring MVC simplifies web development by providing a clean, well-structured approach to handling HTTP requests, processing them, and rendering the appropriate responses.
* It follows the **Model-View-Controller** design pattern, which divides the web application into three core components:

**Model**: Represents the application data or business logic.

**View**: Responsible for rendering the user interface (UI) and displaying data.

**Controller**: Handles the user input, updates the model, and selects the view to display.

* Spring MVC integrates tightly with other features of the Spring Framework, such as **dependency injection**, **transaction management**, and **data access**, making it a powerful and cohesive solution for enterprise-grade applications.

**Spring MVC Components**:

* **Controller**: Handles HTTP requests and returns a response.
* **Model**: Holds the data to be displayed on the view.
* **View**: Renders the data from the model in a user-friendly format (e.g., JSP, Thymeleaf).
* **DispatcherServlet**: Central servlet in Spring MVC that manages the request flow.

**Request Mapping in Spring MVC**:

* Using @RequestMapping, @GetMapping, and @PostMapping annotations to map HTTP requests to controller methods.

### @RequestMapping

* The @RequestMapping annotation is a generic and versatile annotation used to map HTTP requests to specific handler methods in the controller. It can handle all types of HTTP methods (GET, POST, PUT, DELETE, etc.) and can also handle URL patterns, parameters, headers, etc.
* You can use @RequestMapping to map a method to any type of HTTP request (e.g., GET, POST). It's the most general-purpose annotation.

### @GetMapping

* @GetMapping is a specialized version of @RequestMapping that specifically maps HTTP GET requests to a controller method. It's a shortcut for mapping GET requests without explicitly specifying the method type.
* Use @GetMapping when you want to handle HTTP GET requests. It’s more concise and readable than @RequestMapping(method = RequestMethod.GET).

### @PostMapping

* @PostMapping is a specialized version of @RequestMapping that specifically maps HTTP POST requests to a controller method. It’s a shortcut for mapping POST requests.
* Use @PostMapping when you want to handle HTTP POST requests. It is similar to @RequestMapping(method = RequestMethod.POST) but is more concise.
* Path variables, request parameters, and form handling.
* Path variables allow you to extract values directly from the URI path. They are part of the URL and are typically used for identifying resources (e.g., retrieving a specific product by its ID).
* You can define path variables in your @GetMapping, @PostMapping, or @RequestMapping annotations by using {} syntax in the URL pattern.
* If you have a URL like /products/123, the value 123 can be accessed as a path variable in your controller.
* Request parameters are key-value pairs sent in the URL query string (for GET requests) or in the body of the request (for POST requests).
* These parameters are extracted using @RequestParam in the controller methods.
* Form handling refers to processing data submitted by the user through HTML forms. Spring MVC provides a simple way to bind form data to Java objects and process the submission using @ModelAttribute and @RequestParam.