**Spring**

**Spring Container:** It is core part of the spring framework which is responsible for creation, configuration and managing the lifecycle of beans.

**SpringBean Life cycle:**

1. **Bean Instantiation -** Spring instantiates the required classes, and registers the resulting object as a bean in the container.
2. **Populate Properties -** Spring injects dependencies via setter or constructor injection.
3. **Bean Initialization –** Spring performs initialization logic, which may include calling afterPropertiesSet(), if the bean implements InitializingBean.
4. **Bean Ready to Use**
5. **Destruction –** When the application context is closed, destroy() is called if the bean implements DisposableBean, or custom destroy() is called.

**What is dependency in Spring?**

Dependencies are the properties of bean, primitive like int, float, and even non primitive like objects of another class.

**Dependency Injection:**

* Dependency Injection (DI) is a design pattern used in software development, especially in frameworks like Spring, to achieve loose coupling between classes and their dependencies.
* Instead of a class creating its own dependencies (objects it needs to work), those dependencies are provided (injected) from the outside.

**Without Dependency Injection**

**A screen shot of a computer

AI-generated content may be incorrect.**

**With Dependency Injection**

A computer screen shot of a code

AI-generated content may be incorrect.

A computer screen shot of code

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

**@Bean:**

* In Spring, a bean is an object that is managed by the Spring IoC (Inversion of Control) container.
* @Bean tells the Spring container that the method annotated with @Bean will return an object that should be registered as a Spring bean.

**@Qualifier:**

* It helps Spring know which specific bean to inject when there are multiple beans of the same type.
* For example, ***EmailService*** class and ***SmsService*** class implements the ***MessageService*** interface, and ***NotificationManager*** class is dependent on MessageService, then we can add @Qualifier("instancename”) below @Autowired to specify which bean to inject.

**Autowiring:**

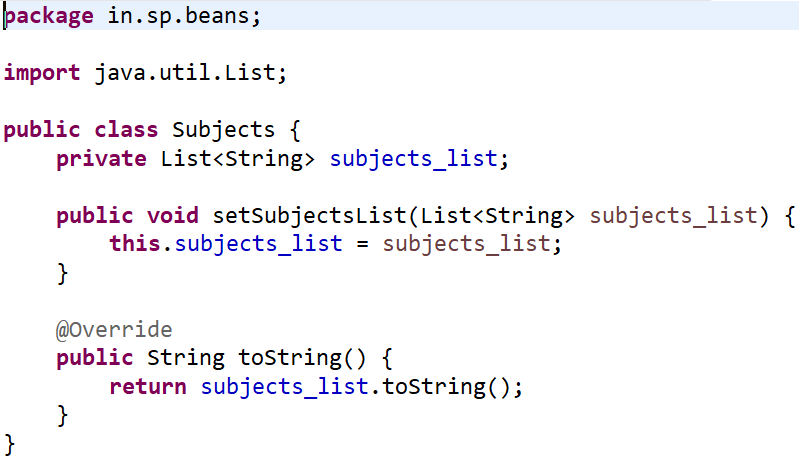
* Autowiring automatically injects dependencies without explicitly specifying them in configuration, using @Autowired annotation.
* Autowiring can only be used with non-primitive types such as Custom classes, as primitive types are not beans and are not managed by Spring Container.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer code

AI-generated content may be incorrect.



A screenshot of a computer program

AI-generated content may be incorrect.

A computer code with text

AI-generated content may be incorrect.

**Injecting Collections in Spring:**

A computer screen shot of code

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

**Inner Beans:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Injecting Interface:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**Spring JDBC:**

1. ***DriverManagerDataSource* class*:***

* It implements *DataSource* interface.
* Used for Database configuration, driver loading and connection creation.

1. ***JdbcTemplate class:***

* Used to perform CRUD operations.
* Some methods are update(), query(), queryForList(), queryForMap(), etc.

**Design Pattern:**

A design pattern is a reusable solution to a commonly occurring problem. It is not a finished design that can be directly transformed into code, but rather a template or blueprint for solving a problem in a way that has been proven effective.

**MVC Design Pattern:**

1. **Model**

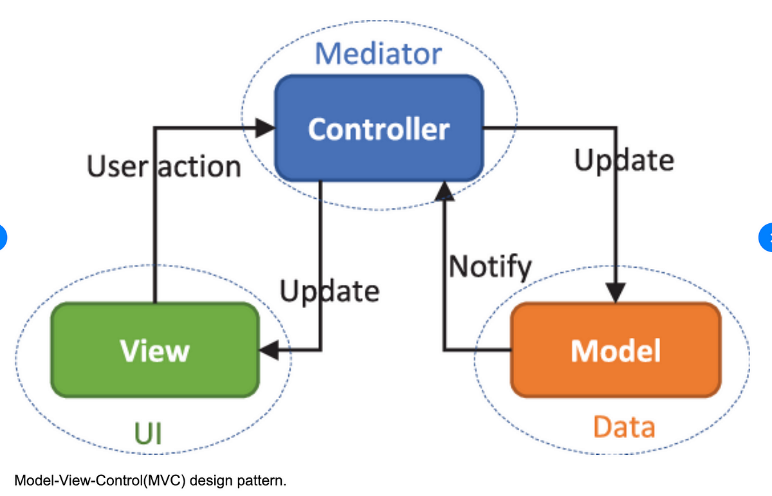
* Represents the data and business logic of the application.
* Interacts with database.

1. **View**

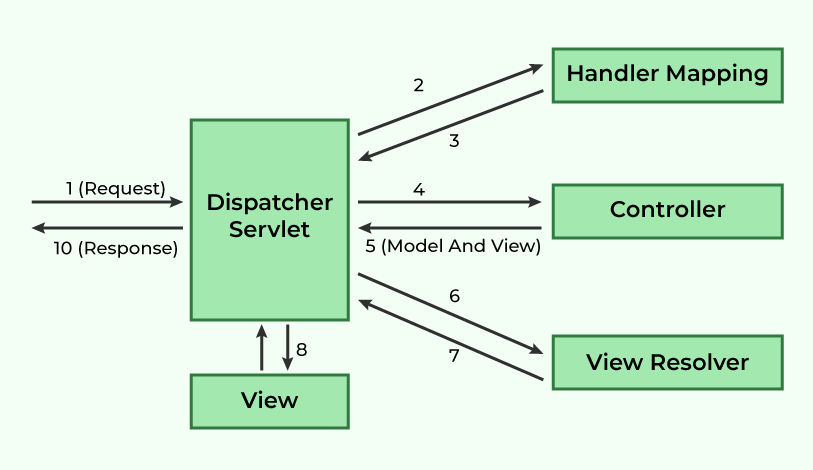
* Represents the user interface.
* Sends the user actions (button clicks) to the controller

1. **Controller**

* Acts as a middleman between the view and model.
* Handles user input, processes it (possibly using the model), and updates the view.



**Spring MVC:**



1. All incoming requests are catched by the DispatcherServlet, which is known as the front controller.
2. The dispatcher servlet checks the HandlerMapping, and finds out which controller should handle the request.
3. The selected controller processes the request, and returns a Model and View object.
4. Dispatcher servlet checks the View Resolver and invokes the appropriate view component.
5. View uses the model data to render the final page, and then response is sent to the client.

**Steps to create Spring MVC Application**

1. **Configure Dispatcher Servlet in web.xml**
2. **Create Spring Configuration File.**
3. **Configure view Resolver**
4. **Create Controllers**
5. **Create a view to show page.**