# What is Spring MVC?

Spring MVC is a module of spring framework that is used to build web applications. It is based on Model View Controller architecture.

- Model: Model contains the data of application

- Controller: Controller contains the business logic of application

- view: View is used to represent data of application in a particular format.

# Explain Flow of Spring MVC?



# Give some advantages of Spring MVC?

* Separate roles
* Light-weight
* Reusable business code
* Flexible Mapping
* Inversion of control
* Dependency Injection

**Controller:** The Controller is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

# Common Annotations used:

**@Controller**: it is used to specify that the class is a controller.

**@RequestMapping:** it is used to map web requests onto specific handler classes/methods.

**@GetMapping, @PostMapping, @PutMapping, @PatchMapping, @DeleteMapping**

**@ResponseBody:** it is used to bound the method return values with the response body. It also converts java objects to json.

(jo hum return karte hai wohi view page pr display ho jata h.)

**@RequestBody:**

* indicates a method parameter should be bound to the body of the web request
* The body of the request is passed through an {@link HttpMessageConverter} to resolve the method argument depending on the content type of the request.
* Optionally, automatic validation can be applied by annotating the argument with {@code @Valid}

**@ModelAttribute:**

- It can be used in two way

1. In method arguments: It binds the form data with a bean.

Ex: public String submit(@ModelAttribute("employee") Employee employee)

2. On method: Spring MVC will call this method before every request handler method.

Ex: @ModelAttribute

public void xyz(Model model)

**@PathVaribale:**It is used to extract the variable from the requested URI. e.g <http://localhost/student/id> . To extract id from this uri.

* indicates that a method parameter should be bound to a URI template variable.

**@RequestParam:** It is used to extract query parameters from request URL.

E.g. <http://localhost/student/getdetails?name=>’ ’& course=’ ’. To extract values of name and course.

* indicates that a method parameter should be bound to a web request parameter.

**@Valid:** It is used to validate the input parameter of entity class.

**@Validated:**It is used to validate parameters in small groups.

**@PropertySource:** used to specify the location of a properties file that contains key-value pairs for configuring properties in a Spring application.

**@Bean:** Indicates that a method produces a bean to be managed by the Spring container.

**@Autowired:** Marks a constructor, field, setter method, or config method as to be autowired by Spring's dependency injection facilities.

**@Value :** Use to provide default value to variables and method arguments.

**@Qualifier:** used to differentiate a bean among the same type of bean objects using a fully qualified name.

**@Component:** used to indicate that a class is a Spring-managed component, allowing it to be automatically discovered and registered in the Spring application context as a bean.

**@Configuration:**. used to declare a class as a configuration class in Spring, indicating that it contains bean definitions and other Spring configuration elements.

**@ComponentScan**: Use with @Configuration to scan the packages that we wanted to be scanned**,@ComponentScan without argument scans the base packages and all it’s sub packages.**

**@Service:** indicate that a class is a "Business Service Facade" (in the Core J2EE patterns sense), or something similar.

This annotation serves as a specialization of {@link Component @Component},

\* allowing for implementation classes to be autodetected through classpath scanning.

**@Scheduled:** Defines a method to be scheduled for periodic execution.

**@Profile:** Specifies that a component should be registered only if a specific profile is active.

**@PostConstruct:** called after the bean has been initialized and before this bean is returned to the requested object.

**@PreDestroy:** is called just before the bean is removed from the container.

**@Transactional:** The @Transactional annotation makes use of the attributes rollbackFor or rollbackForClassName to rollback the transactions, and the attributes noRollbackFor or noRollbackForClassName to avoid rollback on listed exceptions

* The default rollback behavior in the declarative approach will rollback on runtime exceptions.

**@Required:** Marks a method (typically a JavaBean setter method) as being 'required': that is, the setter method must be configured to be dependency-injected with a value.

* if not set it throws BeanInitializationException.

**@Repository:** Indicates that an annotated class is a "Repository", originally defined by Domain-Driven Design (Evans, 2003) as "a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects".

It is used to mark a class that interacts with the database.

this annotation also serves as a specialization of {@link Component @Component}, allowing for implementation classes to be autodetected

through classpath scanning.

**@ControllerAdvice:** allows handling exceptions across the whole application in one global handling component.

**@ExceptionHandler:** Annotation for handling exceptions in specific handler classes and/or handler methods.

**@EnableWebMvc:** Adding this annotation to an {@code @Configuration} class imports the Spring MVC configuration from {@link WebMvcConfigurationSupport}.

only one {@code @Configuration} class may have the @EnableWebMvc annotation to import the Spring Web MVC configuration.

# What is the difference between PUT, PATCH, POST?

* Use POST when you want to create a new resource.
* Use PUT when you want to update or create a resource with a complete representation.
* Use PATCH when you want to apply partial updates to a resource.

# Explain following classes/interfaces or terms:

**Model:** The Model class is used to pass data from the controller to the view in the MVC pattern.

**ModelAndView:** It is a class that combines both model data and view information in a single object.

**BindingResult:** It is an interface which is used for handling validation and binding errors during form submissions.

**Object Mapper**: It is used to serialize and deserialize java objects.

**ResponseEntityExceptionHandler:**  A class with an {@code @ExceptionHandler} method that handles all Spring MVC raised exceptions by returning a {@link ResponseEntity}.

Convenient as a base class of an {@link ControllerAdvice @ControllerAdvice}

\* for global exception handling in an application.

**WebMvcConfigurer:** Defines callback methods to customize the Java-based configuration for Spring MVC enabled via {@code @EnableWebMvc}.

code @EnableWebMvc -annotated configuration classes may implement

this interface to be called back and given a chance to customize the

default configuration.

**RestTemplate:**  Synchronous client to perform HTTP requests, exposing a simple, template method API over underlying HTTP client libraries such as the JDK

HttpURLConnection , Apache HttpComponents, and others.

It is used to establish communication between two api’s.

**HandlerInterceptor:** Workflow interface that allows for customized handler execution chains. Applications can register any number of existing or custom interceptors for certain groups of handlers, to add common preprocessing behavior without needing to modify each handler implementation.

A HandlerInterceptor gets called before the appropriate HandlerAdapter

triggers the execution of the handler itself. This mechanism can be used

for a large field of preprocessing aspects, e.g. for authorization checks.

**SecurityFilterChain:** Defines a filter chain which is capable of being matched against an HttpServletRequest. in order to decide whether it applies to that request.

**PasswordEncoder:** Service interface for encoding passwords.

The preferred implementation is {@code BCryptPasswordEncoder}.

**UserDetailService:** Core interface which loads user-specific data.

It is used throughout the framework as a user DAO and is the strategy used by the org.springframework.security.authentication.dao.DaoAuthenticationProvider

DaoAuthenticationProvider}.

**Authentication Provider:** Performs authentication with the same contract as

{@link org.springframework.security.authentication.AuthenticationManager#authenticate(Authentication)}

@param authentication the authentication request object.

@return a fully authenticated object including credentials. May return null if the AuthenticationProvider is unable to support

authentication of the passed Authentication object. In such a case, the next AuthenticationProvider that supports the presented Authentication class will be tried.

@throws AuthenticationException if authentication fails.

**Authentication Manager:** Attempts to authenticate the passed {@link Authentication} object, returning a fully populated Authentication object (including granted authorities) if successful.

**Jwts:** Factory class useful for creating instances of JWT interfaces. Using this factory class can be a good alternative to tightly coupling your code to implementation classes.

**Claims:** A JWT Claims set.

This is ultimately a JSON map and any values can be added to it, but JWT standard names are provided as type-safe getters and setters for convenience.

# Other Annotation:

**@JsonProperty:** The @JsonProperty annotation is used to specify the property name in a JSON object when serializing or deserializing a Java object using the Jackson library

**@AllArgsConstructor:** generates a constructor that initializes all fields in the class.

**@NoArgsConstructor :** generates a constructor with no parameters, initializing all fields with default values.

**@Data :** This annotation generates the [@ToString](https://projectlombok.org/features/ToString), [@EqualsAndHashCode](https://projectlombok.org/features/EqualsAndHashCode), [@Getter / @Setter](https://projectlombok.org/features/GetterSetter) and [@RequiredArgsConstructor](https://projectlombok.org/features/constructor) together for annotated class fields**.**

# JPA Annotations :

**@Table :** It is used to specify the details of the table, by default provide the name same as the class name in the database.

**@Entity:** It is used to annotate that this class is an entity class.(Entities are nothing but POJOs, used to represent tables stored in a database.)

@Embaddable: Annotation use to notify that this class will be going to embedded by another entity.

@Embedded: Use to embedded the class inside an entity.

**@Id:** It is used to define the primary key.

**@GeneratedValue**

**@Column :** Use to add the Column in the table with the provided name ,else will by default provide the name same as field name.

**@Transient :** ignores the fields data to be inserted in the database.

**Scheduler:**

**@Scheduled(fixedDelay = 1000 )**

**@Scheduled(fixedRate = 1000 )**

**@Scheduled(cron = "0 15 10 15 \* ?")**

**Q:- What is circular dependency in spring?**

**A circular dependency occurs when a bean A depends on another bean B, and the bean B depends on bean A as well:**

**Bean A → Bean B → Bean A**

**Of course, we could have more beans implied:**

**Bean A → Bean B → Bean C → Bean D → Bean E → Bean A**

**It can happen in Spring when using constructor injection.**

**If we use other types of injections, we shouldn't have this problem since the dependencies will be injected when they are needed and not on the context loading.**

**A simple way to break the cycle is by telling Spring to initialize one of the beans lazily. So, instead of fully initializing the bean, it will create a proxy to inject it into the other bean. The injected bean will only be fully created when it’s first needed.**

**@Component**

**public class CircularDependencyA {**

**private CircularDependencyB circB;**

**@Autowired**

**public CircularDependencyA(@Lazy CircularDependencyB circB) {**

**this.circB = circB;**

**}**

**}**

**Maven command**

**Git commands**

**Creating docker file**

**Docker File-**

FROM openjdk:8

EXPOSE 8080

ADD spring-mvc-docker.jar spring-mvc-docker.jar

ENTRYPOINT [ "java","-jar","/spring-mvc-docker.jar" ]

**Docker-compose file -**

**version: '3.8'**

**services:**

**mysqldb:**

**image: mysql**

**container\_name: mysql**

**ports:**

**- '3306:3306'**

**environment:**

**MYSQL\_ROOT\_PASSWORD: root**

**MYSQL\_DATABASE: mydb**

**MYSQL\_USER: myuser**

**MYSQL\_PASSWORD: mypassword**

**volumes:**

**- dbdata:/var/lib/mysql**

**volumes:**

**dbdata:**

**Q:Purpose of rt.jar:**

**In Java, `rt.jar` stands for "RunTime JAR." It used to be a critical part of the Java Runtime Environment (JRE) and contained many of the core classes and libraries required for running Java applications. However, it's important to note that the concept of `rt.jar` has evolved over time, and its structure has changed with different Java versions.**

**As of Java 9, the structure of the JRE changed significantly with the introduction of the module system (Java Platform Module System, or JPMS). One of the major changes was the modularization of the JDK, which included breaking down the monolithic `rt.jar` into a collection of smaller, more manageable modules. This was done to improve modularity, maintainability, and security.**

**Here are some key points related to `rt.jar`:**

**1. \*\*Pre-Java 9\*\*: In Java versions prior to Java 9, `rt.jar` was a single JAR file containing the core classes of the Java Standard Library. It was present in the `lib` directory of the JRE installation and was loaded by the classloader at runtime. It included classes like `java.lang.Object`, `java.lang.String`, and many others used by every Java application.**

**2. \*\*Java 9 and Beyond\*\*: With the introduction of the module system in Java 9, the contents of the JRE were modularized. Core classes and APIs were organized into distinct modules, and `rt.jar` was split into multiple JAR files, each representing a separate module. This modularization aimed to improve security, performance, and maintainability.**

**The concept of the "classpath" also evolved with modules. Instead of relying on a single monolithic `rt.jar`, developers now work with modules that declare their dependencies explicitly.**

**3. \*\*Obsolete in Java 11+\*\*: Starting from Java 11, Oracle's OpenJDK distribution and other compatible JDK distributions do not provide `rt.jar` anymore. Instead, the core Java classes are organized into separate modules, and the classpath structure has changed to align with the module system.**

**In summary, `rt.jar` was a significant component in pre-Java 9 Java Runtime Environments, containing core classes and libraries. However, with the modularization introduced in Java 9, the way Java classes and libraries are organized and loaded has changed, making `rt.jar` as it was known obsolete in more recent Java versions.**

**Hystrix in spring**

**To respond with a particular fallback method if one or more of the microservices are down in an application.**

**Hibernate:**

Hibernate is java framework use to interact java application with database.

-It is a ORM (Object relational Mapping)tool which is use to map our java object with data stored in database.

Hibernate Architecture -



-