1. Can you explain the components of the MVC architecture in Spring?

In Spring MVC, the architecture is based on the Model-View-Controller design pattern. The **Model** represents the application's data and business logic, typically encapsulated in Java classes and managed using services and DAOs. The **View** is responsible for rendering the response to the client—most often a JSP or a frontend template engine like Thymeleaf. Finally, the **Controller** is the component that handles incoming HTTP requests, interacts with the model to process data, and returns the appropriate view name. The DispatcherServlet acts as the front controller, delegating the request to the right handler (controller), and coordinating the flow between model and view.

2. How is the Model, ModelMap, and ModelAndView used in Spring?

In Spring MVC, `Model`, `ModelMap`, and `ModelAndView` are used to pass data from the controller to the view. The `Model` interface allows you to add attributes to the model, which can then be accessed in the view layer. `ModelMap` is similar but allows for chainable calls and gives access to a Map-style structure. `ModelAndView` combines both the model and view in a single return object, where you can set the view name and add model attributes. All three essentially help in binding and transporting data to the view, but `ModelAndView` is typically used when you want to explicitly define both model and view in one object.

3. What is the difference between Model and ModelAttribute in Spring MVC?

The `Model` is an interface used to define a holder for model attributes passed to the view, while `@ModelAttribute` is an annotation used to bind a method parameter or method return value to a named model attribute. When applied to a method parameter, `@ModelAttribute` helps to bind incoming request data to the object, which is then automatically added to the model. It is also used to pre-populate model attributes before any handler method is invoked. So, `Model` is more of a container, whereas `@ModelAttribute` is a mechanism to bind and manage data flow between request and model.

4. What is the use of the View Resolver?

The View Resolver in Spring MVC is responsible for resolving logical view names returned by controllers into actual view resources. For example, if a controller returns "home" as the view name, the view resolver will map this to something like `/WEB-INF/views/home.jsp` depending on the configuration. It abstracts the physical view technology from the controller logic and supports pluggable view technologies like JSP, Thymeleaf, and FreeMarker.

5. Validation for Employee with id, name, email, salary, phoneNo, and age in Spring MVC?

To perform validation in Spring MVC for an Employee object with fields like id, name, email, salary, phone number, and age, you typically use JSR-303 Bean Validation annotations like `@NotNull`, `@Size`, `@Email`, `@Min`, `@Max`, and so on within the Employee class. The controller method handling the form would use `@Valid` on the `@ModelAttribute`, and you'd capture binding results using a `BindingResult` parameter. For example, `@Valid @ModelAttribute("employee") Employee employee, BindingResult result` allows Spring to validate the object and collect any validation errors, which you can then handle and display in the view.

6. What is DispatcherServlet and what role does it play in Spring MVC?

The `DispatcherServlet` is the central component of the Spring MVC framework. It acts as the front controller that intercepts all incoming HTTP requests and dispatches them to the appropriate controller methods. It handles the entire request processing lifecycle, including invoking the handler mappings, invoking the appropriate controller, calling the view resolver, and finally rendering the view. It's configured in the web application and is the entry point for any Spring MVC-based web application.

7. How does Spring handle form validation and error binding?

Spring handles form validation by integrating with the Bean Validation API (JSR-303) through annotations like `@NotEmpty`, `@Email`, `@Min`, etc., on the model object. In the controller, you annotate the model object with `@Valid` and use a `BindingResult` parameter immediately after it to capture validation errors. If there are errors, they can be checked using `result.hasErrors()`, and appropriate logic can be applied, such as returning the form view again with error messages. These errors can be displayed in the view using Spring's form tags like `<form:errors>`.

8. How do you handle custom validators in Spring?

To create a custom validator in Spring, you implement the `Validator` interface or use the `@Constraint` annotation to define a custom constraint for use with JSR-303. You then override the `supports()` and `validate()` methods to define the validation logic. The custom validator can be registered using `@InitBinder` in the controller or be wired in using annotations if it's a JSR-303 validator. This allows you to handle complex validation scenarios that go beyond standard annotations.

9. What annotations are used to bind form data to model attributes?

In Spring MVC, the primary annotation used to bind form data to model attributes is `@ModelAttribute`. This annotation tells Spring to populate the object with form data from the HTTP request. Additionally, `@RequestParam` can be used for individual request parameters, and `@RequestBody` is used for binding JSON or XML payloads in RESTful services. However, for traditional form submissions in web apps, `@ModelAttribute` is most commonly used.

10. How does Spring Boot simplify Spring application configuration?

Spring Boot simplifies Spring application development by eliminating much of the boilerplate configuration required in traditional Spring applications. It uses **convention over configuration**, embedded servers like Tomcat, auto-configuration based on classpath dependencies, and production-ready features like health checks and metrics through Actuator. Instead of writing extensive XML or Java-based configuration, developers can just include the right dependencies and Spring Boot will intelligently auto-configure most of the components. It also provides a simple way to externalize configurations via `application.properties` or `application.yml`.

11. How does request mapping work in Spring MVC and what is the use of @RequestMapping annotation?

In Spring MVC, request mapping is the mechanism that maps incoming HTTP requests to appropriate handler methods in controller classes. This is primarily done using the @RequestMapping annotation, which can be applied at both class and method levels. It allows developers to specify the URL pattern, HTTP method (GET, POST, etc.), headers, and more. When a request comes in, the DispatcherServlet scans the registered controllers and invokes the method whose mapping matches the request URL and method. This makes the controller methods more readable and maintainable by clearly associating them with specific routes.

12. What is the difference between @RequestParam and @PathVariable in Spring MVC?

The difference between @RequestParam and @PathVariable lies in how the parameters are passed and retrieved in a controller. @RequestParam is used to extract query parameters from the URL, such as /search?name=John, where “name” is accessed via @RequestParam("name"). On the other hand, @PathVariable is used to extract values from the URI path itself, like /user/123, where “123” can be accessed with @PathVariable("id"). Essentially, @RequestParam deals with key-value pairs in the query string, while @PathVariable works with dynamic URI segments.

13. **How can you handle exceptions globally in a Spring MVC application?**

    Exception handling in Spring MVC can be done globally using the @ControllerAdvice annotation combined with @ExceptionHandler methods. This approach allows a centralized class to catch exceptions thrown by any controller and return appropriate responses or views. It's useful for handling common errors like validation failures, database exceptions, or any custom business exceptions. This improves code reusability and keeps controllers clean by removing repetitive try-catch blocks.

14. **What are interceptors in Spring MVC and how are they different from filters?**

    Interceptors in Spring MVC are used to intercept requests before they reach the controller and after the controller returns the response. They are configured using the HandlerInterceptor interface and are often used for cross-cutting concerns like logging, authentication checks, and performance monitoring. Filters, in contrast, are part of the servlet specification and operate at a lower level, even before Spring comes into play. While filters can work on any kind of request, interceptors are specific to Spring MVC and have access to Spring-specific objects like the handler and model.

15. **Explain the lifecycle of a Spring MVC request. What happens from the moment a request hits the DispatcherServlet?**

    The lifecycle of a Spring MVC request begins when the request is received by the DispatcherServlet, which acts as the front controller. It consults the HandlerMapping to determine the correct controller to handle the request. Once the controller method is invoked, it processes the request, interacts with the service and data layers if needed, and returns a logical view name and model data. The DispatcherServlet then passes this to the ViewResolver, which maps the view name to an actual view (like a JSP or Thymeleaf template). Finally, the view is rendered and the response is sent back to the client.

16. **What is the role of HandlerMapping and HandlerAdapter in Spring MVC?**

    HandlerMapping and HandlerAdapter are internal components that support the DispatcherServlet in processing requests. HandlerMapping is responsible for mapping incoming requests to appropriate handler methods, based on the URL, HTTP method, and other attributes. Once the handler is found, the DispatcherServlet uses HandlerAdapter to invoke the handler method. The adapter acts as a bridge between the DispatcherServlet and various handler types, making it possible to support different types of controllers.

17. **How do you upload and handle file uploads in Spring MVC?**

    File uploads in Spring MVC can be handled using the MultipartFile interface. To support file uploads, the application must be configured with a MultipartResolver bean, such as CommonsMultipartResolver. In the controller,

you can define a method that accepts a MultipartFile parameter, which represents the uploaded file. This file can be saved to the server or processed as needed. Spring also provides validation and size-limiting options for uploaded files.

18. **Can you explain the difference between Forward and Redirect in Spring MVC and how to implement them?**

    In Spring MVC, the terms Forward and Redirect refer to two ways of transferring control to another resource. A forward is handled internally by the server without changing the URL in the browser; it's typically used when processing and displaying data within the same request context. A redirect, on the other hand, tells the client to make a new request to a different URL, which changes the browser's address bar. You can forward by returning a view name like "forward:/home" and redirect using "redirect:/login" in the controller.

19. **How do you secure a Spring MVC application using Spring Security?**

    Securing a Spring MVC application is typically done using Spring Security. It integrates seamlessly with the MVC framework and allows developers to define security rules for authentication and authorization. You can configure access controls for URLs, roles, and methods using annotations like @PreAuthorize or XML/Java-based security configurations. Spring Security also provides features like form-based login, password encryption, session management, and CSRF protection, making it a comprehensive solution for securing web applications.

20. **What is content negotiation in Spring MVC and how is it configured?**

    Content negotiation in Spring MVC refers to the ability to return different representations of the same resource, such as JSON, XML, or HTML, depending on the client's request. Spring achieves this using the ContentNegotiationManager and various HttpMessageConverter implementations. The client can specify the desired response type using the Accept header, and Spring will choose the appropriate converter. This is especially useful in RESTful applications where clients may need different formats of the same data.