1. **What is the DispatcherServlet in Spring Boot?**

    - In Spring MVC the frontend controller is “Dispatcherservelet” which needs to configured in web.xml file.
    - It is a central component that handles all incoming HTTP requests and maps them to appropriate controllers.
    - Dispatcherservlet loads the spring bean XML configuration file.
    - It is a part of Spring Web MVC framework and acts as entry point for all requests.

2. **Explain the life cycle of a Spring Bean?**

    The lifecycle of a Spring Bean involves several stages, from creation to destruction.

    1. **Instantiation:**
        - The Spring container creates an instance of the bean using reflection. The bean can be defined in either XML configuration, Java-based configuration (using @Configuration), or annotations (e.g., @Component).
        - The constructor is called, and dependencies are injected.
    2. **Populate Properties:**
        - Spring injects dependencies into the bean. These dependencies could be other beans or values specified in the configuration.
        - Spring can inject these properties using Constructor injection, Setter injection, Field injection.
    3. **BeanNameAware Interface:**
        - If the bean implements BeanNameAware, the Spring container calls its setBeanName() method, passing the bean's ID defined in the configuration.
        - This allows the bean to be aware of its own name in the Spring context.
    4. **ApplicationContextAware Interface:**
        - If the bean implements BeanFactoryAware or ApplicationContextAware, Spring injects the BeanFactory or ApplicationContext respectively.
        - This allows the bean to be aware of the context or the factory managing it.
    5. **Pre-Initialization:**
        - Before the bean’s initialization method is called, Spring will pass it through any BeanPostProcessor that is registered.
        - BeanPostProcessors allow you to modify the bean or perform some actions before the bean is fully initialized.
        - The postProcessBeforeInitialization() method of the BeanPostProcessor interface is called here.
    6. **InitializingBean Interface:**
        - If the bean implements the InitializingBean interface, the afterPropertiesSet() method will be called.
    7. Alternatively, if a custom initialization method is defined in the bean configuration (via @Bean(initMethod="initMethodName") or in XML configuration), it will be invoked at this stage.

8. **Post-Initialization:**
    - After the initialization methods are invoked, the bean is passed again through any BeanPostProcessor.
    - The postProcessAfterInitialization() method of the BeanPostProcessor interface is called here.
9. **Bean Ready for Use:**
    - The bean is now fully initialized and ready for use in the Spring context. It can now be injected into other beans or accessed through the container.
10. **Destruction:**
    - When the Spring container shuts down or when the bean's scope is limited (like in prototype scope), the destruction lifecycle begins.
    - If the bean implements DisposableBean, its destroy() method is called.
    - Additionally, a custom destroy method can be specified in the configuration (@Bean(destroyMethod="destroyMethodName") or in XML).
    - If the bean has any @PreDestroy annotation, it will also be invoked before the bean is destroyed.

3. What is the difference between @Controller and @RestController?

1. **RestController:**

    - A convenience annotation for building RESTful web services.
    - Designed for controllers that return data (typically JSON/XML) instead of views.
    - It is a shortcut for @Controller + @ResponseBody.
    - Return values are automatically serialized to JSON/XML, and written directly to the HTTP response body.

2. **Controller:**

    - Used in traditional MVC (Model-View-Controller) architecture.
    - Typically returns a view (JSP, Thymeleaf template) and not raw data.
    - Return values from methods are interpreted as view names unless annotated with @ResponseBody.
    - To return data (JSON or plain text), you must annotate methods with @ResponseBody.

4. How do you secure a REST API in Spring Boot using Spring Security?

- Spring Security provides a powerful and customizable authentication and access-control framework to secure REST APIs.
- In a typical setup, you define security rules in a configuration class using HttpSecurity, where you specify which endpoints are public and which require authentication.

- Authentication mechanisms like Basic Authentication, JWT (JSON Web Tokens), or OAuth2 can be used to verify user identity. For stateless REST APIs, CSRF protection is usually disabled, and sessions are not used, making the API stateless and secure.

5. **What is the significance of application.yml over application.properties?**

   The application.yml file and the application.properties file in Spring Boot serve the same purpose: external configuration of your application (like ports, database settings, logging, etc.). However, application.yml offers more powerful and cleaner configuration, especially for complex or hierarchical data.

   application.yaml over application.properties:

   - YAML uses indentation to represent nested data, making it cleaner and easier to read.
   - Ideal for hierarchical configurations like multiple data sources or Spring profiles.
   - More intuitive for arrays, maps, and complex structures.
   - You can define multiple profiles in one YAML file using --- separators.
   - Less repetitive syntax compared to key-value pairs in application.properties.

6. **(MCQ) Which layer should contain the business logic in a Spring Boot application?**
   a)Controller
   b)Service
   c)Repository
   d)Entity

   - **b) Service**

7. **Explain how @Transactional works in Spring Boot.**

   - The @Transactional annotation in Spring Boot is used to manage transactions declaratively, typically in the Service layer. It ensures that a method runs within a database transaction, so that all operations within it either complete successfully together or fail together (rollback).
   - All operations inside the method annotated with @Transactional are treated as a single unit of work.

8. **What is Lazy Loading vs Eager Loading in JPA?**

   - In JPA, Lazy Loading and Eager Loading determine when related entities are fetched from the database.
   - Lazy Loading means that related data is not loaded immediately with the parent entity; instead, it is loaded only when accessed for the first time. This approach optimizes performance by avoiding unnecessary database queries

and loading data only when needed. However, it can cause issues like LazyInitializationException if the related data is accessed outside an active persistence context.

- On the other hand, Eager Loading means that related entities are fetched immediately along with the parent entity during the initial database query. This ensures that all required data is available upfront, but it can lead to performance degradation if large amounts of data are fetched unnecessarily or if it triggers multiple joins.
- lazy loading is preferred for large or optional relationships, while eager loading is suitable when related data is always required.

9. **How can you upload files using Spring Boot REST APIs?**

- To upload files in a Spring Boot REST API, you create a controller endpoint that accepts a file sent as part of a multipart HTTP request.
- Spring Boot provides the MultipartFile interface, which represents the uploaded file and allows easy access to its contents, name, and metadata.
- In the controller method, you annotate a parameter with @RequestParam("file") of type MultipartFile to receive the file. Inside the method, you can process or save the file to a local directory, database, or cloud storage.
- Spring Boot automatically handles multipart request parsing, and you can configure maximum file size limits in application properties.
- Clients send files using POST requests with content type multipart/form-data, attaching the file under the specified parameter name.

10. **What is the purpose of the CommandLineRunner interface?**

The CommandLineRunner interface in Spring Boot is used to execute code after the Spring Boot application has started and the application context is loaded. It provides a single method, run(String... args), where you can put initialization logic, such as setting up default data, running startup tasks, or performing any action that needs to happen once the app is ready.