REVATURE

Spring All Modules

CLIENT INTERVIEW FAQ V1.0

REVATURE

Table of Contents

General Spring

Q1. Have you used & worked on Spring? Tell me about your Spring knowledge?

Answer: Yes, we have learned various Spring modules like, Core Spring, Spring MVC, Spring Data JPA, Spring AOP, Spring Boot etc. We have implemented the RESTful microservices final capstone project as well using Spring boot.

Q2. What is Framework?

Answer: A framework generally provides some base functionality which can be used and extend to make complex applications easily. E.g., Spring MVC It provides everything you need to get off the ground building website using the MVC pattern, it handles web requests, view, routes etc.

Explanation: A software framework, in computer programming, is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality. Frameworks are a special case of software libraries in that they are reusable abstractions of code wrapped in a well-defined Application programming interface (API), yet they contain some key distinguishing features that separate them from normal libraries.

Q3. What is Spring Framework?

Answer: Spring is award winning dependency injection framework used to build simple, web & complex enterprise applications more easily with the set of predefined modules.

Explanation: The Spring framework is one of the most popular application development frameworks of Java. The important feature of the spring framework is dependency injection or the Inversion of Control. With the help of Spring Framework, we are able to develop a loosely coupled application. It's packed with some nice out of the box modules like:

1. Spring JDBC
2. Spring MVC
3. Spring Security
4. Spring AOP
5. Spring ORM
6. Spring Test

These modules can drastically reduce the development time of an application. For example, in the early days of Java web development, we needed to write a lot of boilerplate code to insert a record into a data source. By using the **JdbcTemplate** of the Spring JDBC module, we can reduce it to a few lines of code with only a few configurations.

Q4. What are various projects offered by Spring? List few of them.

Answer: Spring is modular by design. From configuration to security, web apps to big data—whatever the infrastructure needs of your application may be, there is a Spring Project to help you build it. Few commonly used projects listed below-

1. **Spring Framework** - Provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.
2. **Spring Boot** - Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.
3. **Spring Data** - Provides a consistent approach to data access – relational, non-relational, map-reduce, and beyond.
4. **Spring Cloud** - Provides a set of tools for common patterns in distributed systems. Useful for building and deploying microservices.
5. **Spring Security** - Protects your application with comprehensive and extensible authentication and authorization support.
6. **Spring Batch** - Simplifies and optimizes the work of processing high-volume batch operations.
7. **Spring HATEOAS** - Simplifies creating REST representations that follow the HATEOAS principle.
8. **Spring REST Docs** - Lets you document RESTful services by combining hand-written documentation with auto-generated snippets produced with Spring MVC Test or REST Assured.
9. **Spring For Apache Kafka** - Provides Familiar Spring Abstractions for Apache Kafka.
10. **Spring Vault** - Provides familiar Spring abstractions for HashiCorp Vault

Note: We use first four project as part of our Spring and Microservices curriculum.

Core Spring

Q1. What is Dependency Injection & Where all Spring framework can be used?

Answer: Dependency injection is basically providing the instance variables (referred do as dependencies) that an object needs, instead of having it constructed by the class of that object itself. Spring framework can be used to build Java CUI, GUI, Web, Distributed, Enterprise Applications, as all application types are candidate for Dependency injection.

Explanation: There are scenarios where one java class comprises of dependencies i.e. object of another class. This is particularly called as **Has-A** relationship in java world. Whenever in any java application you have one or more dependencies which you don't want to manage on your own rather want to delegate it to some third-party framework that is right place where Spring framework can be leveraged. The wrong myth for Spring is, it is only used for enterprise application but in reality, it can be used in any type of application as far as you have dependencies in your code. Hence Spring framework can be used in any Core Java, Web, Distributed, Web Services as well as Android application as applicable.

Laptop class having Processor dependency

```
1.  public class Laptop {
2.    private String modelName;
3.    private double price;
```

```
4.    private Processor processor;
5.    // Constructor auto-generated constructor stub
6.    // Getter, Setter auto-generated stub
7.  }
8.
9.  public class Processor {
10.   private String modelName;
11.   private String cacheMemory;
12.   private double price;
13.   // Constructor auto-generated constructor stub
14.   // Getter, Setter auto-generated stub
15. }
16.
17. public class Intel extends Processor {
18.   private String numberOfCores;
19.   // Constructor auto-generated constructor stub
20.   // Getter, Setter auto-generated stub
21. }
22.
23. public class Amd extends Processor {
24.   private boolean graphicsAcceleration;
25.   // Constructor auto-generated constructor stub
26.   // Getter, Setter auto-generated stub
27. }
```

Q2. What is IoC and Dependency Injection in Spring?

Answer: Inversion of control is a generic term and implemented in several ways (events, delegates, service locator & Dependency Injection). DI is a subtype of IOC and is implemented by constructor injection, setter injection or method injection.

Explanation: **IoC** which stands for **Inversion of Control** is also known as **Dependency Injection (DI)** usually referred as **Spring IoC/DI**. Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. In contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code. It is a process whereby objects define their dependencies (that is, the other objects they work with) through constructor arguments (called Constructor based DI) or properties that are set on the object instance after it is constructed (called Setter based DI). The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies.

Note: Remember Dependency Injection (DI) is broader term used and supported by many frameworks including frontend framework like Angular. In Angular we have Services that share data between various components are also configured using DI.

Q3. What is IoC Container or DI Container? (What is the role of an IoC container?)

Answer: In spring context both IoC & DI containers are the same. In Spring the business components are called as Beans. The IoC container manages the lifecycle of Beans.

Explanation: The terms Dependency Injection (DI) & Inversion of Control (IoC) are generally used interchangeably, hence some people say IoC Container and some people say DI container but both terms indicate the same thing. The IoC container that is also known as a DI Container is a framework for implementing automatic dependency injection very effectively. It manages the complete object creation and its lifecycle, as well as it also injects the dependencies into the classes. In the Spring framework, the interface **ApplicationContext** represents the IoC container. The Spring container is responsible for instantiating, configuring and assembling objects known as beans, as well as managing their life cycles. The Spring framework provides several implementations of the **ApplicationContext** interface: **ClassPathXmlApplicationContext** and **FileSystemXmlApplicationContext** for standalone applications, and **WebApplicationContext** for web applications.

Q4. Any DI frameworks apart from Spring?

Answer: Libraries and Frameworks that implement DI

- Spring (Java)
- Google Guice (Java)
- Dagger (Java and Android)
- Castle Windsor (.NET)
- Unity(.NET)

Q5. What are Spring Beans?

Answer: Core business component defined insider Spring applications is termed as **Bean.** Each technology has branded their core business components by certain nomenclature. For example, Business component in JavaEE applications are termed as **Servlet**, in web services they are named as **Resource** in Struts framework they are called **ActionForm** etc.

Q5. What are the ways through which the Spring Beans are configured??

Answer: There are broadly two ways in which Spring beans are configured in application-

1. Using XML configuration – We usually define xml file with standard name as **applicationContext.xml** inside **src/main/resources** folder of your maven project.
2. Using Annotation configuration - **@Bean, @Component, @Service, @Repository, @Controller, @RestController**

Q5. What are the types of dependency injection in Spring?

Answer: Dependency Injection in Spring can be done through **Constructors, Setters or Fields**.
Explanation: Lets understand with example:

Class TextEditor having Validator dependency

```
1.  public class TextEditor {
2.    private Validator;
3.    public TextEditor(Validator validator) {
4.      this.validator = validator;
5.    }
6.    // Constructor auto-generated constructor stub
7.    // Getter, Setter auto-generated stub
8.  }
9.
10. public class GrammerValidator extends Validator {
11.   // Constructor auto-generated constructor stub
12.   // Getter, Setter auto-generated stub
13. }
```

Constructor-Based Dependency Injection - In the case of constructor-based dependency injection, the DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.
Spring resolves each argument primarily by type, followed by name of the attribute, and index for disambiguation.

Constructor-Based Dependency Injection

```
1.  //Annotation Configuration:
2.  @Configuration
3.  public class AppConfig {
4.      @Bean
5.      public Validator grammerValidator(){
6.          return new GrammerValidator();
7.      }
8.
9.      @Bean
10.     public TextEditor textEditor() {
11.         return new TextEditor(grammerValidator());
12.     }
13. }
14. //Or
15. //XML Configuration:
16. <bean id="grammerValidator" class="com.revature.model.GrammerValidator" />
17. <bean id="textEditor" class="com.revature.model.TextEditor">
18.     <constructor-arg type="Validator" index="0" name="validator" ref="grammerValidator"/>
19. </bean>
```

Setter-Based Dependency Injection - In the case of setter-based dependency injection, the container will call setter methods of our class after invoking a no-argument constructor (or no-argument static factory method) to instantiate the bean. We can combine constructor-based and setter-based types of injection for the same bean. The Spring documentation recommends using constructor-based injection for mandatory dependencies, and setter-based injection for optional ones.

Setter-Based Dependency Injection

```
1.  //Annotation Configuration:
2.  @Configuration
3.  public class AppConfig {
4.    @Bean
5.    public TextEditor textEditor() {
6.      TextEditor = new TextEditor();
7.      textEditor.setValidator(grammerValidator());
8.      return textEditor;
```

```
    }
    @Bean
    public Validator grammerValidator() {
      return new GrammerValidator();
    }
}
//Or
//XML Configuration:
<bean id="grammerValidator" class="com.revature.model.GrammerValidator"/>
<bean id="textEditor" class="com.revature.model.TextEditor">
     <property name="validator" ref="grammerValidator"/>
</bean>
```

Field-Based Dependency Injection - In the case of setter-based dependency injection, we inject the dependencies by marking them with `@Autowired` annotation.

This approach might look simpler and cleaner, but it has a few drawbacks such as:

- This method uses reflection to inject the dependencies, which is costlier than constructor-based or setter-based injection.
- It's really easy to keep adding multiple dependencies using this approach. If we were using constructor injection, having multiple arguments would make us think that the class does more than one thing, which can violate the Single Responsibility Principle.

Field-Based Dependency Injection

```
//Annotation Configuration:
//While constructing the TextEditor object,
//if there's no constructor or setter method to inject the Validator bean,
//the container will use reflection to inject Validator into TextEditor.

public class TextEditor{
     @Autowired
     private Validator;
}

//Or
//XML Configuration:

<bean id="grammerValidator" class="com.revature.model.GrammerValidator" />
<bean id="textEditor" class="com.revature.model.TextEditor" autowire="byType"/>
<bean id="textEditor" class="com.revature.model.TextEditor" autowire="byName"/>
```

When using XML-based configuration metadata, we can specify the `autowire` mode for a bean definition with the autowire attribute of the `<bean/>` element. The autowiring functionality has four modes. There are four modes of autowiring a bean using an XML configuration:

1. `no`: The default value – this means no autowiring is used for the bean and we have to explicitly name the dependencies. The bean references must be defined by `ref` elements.
2. `byName`: Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired.
3. `byType`: Lets a property be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use byType autowiring for that bean. If there are no matching beans, nothing happens (the property is not set).

4. **constructor**: Analogous to byType but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

Q6. What are the bean scopes?

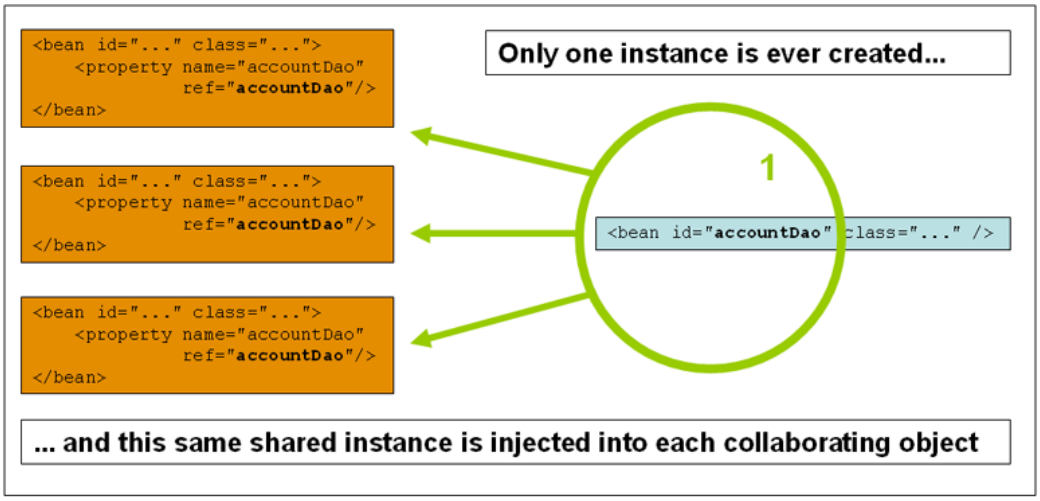
Answer: The scope of a bean defines the life cycle and visibility of that bean in the contexts we use it.

Explanation: You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition but also control the scope of the objects created from a particular bean definition. The latest version of the Spring framework defines 6 types of scopes:

1. **singleton**: (Default) Scopes a single bean definition to a single object instance for each Spring IoC container. When we define a bean with the singleton scope, the container creates a single instance of that bean; all requests for that bean name will return the same object, which is cached. Any modifications to the object will be reflected in all references to the bean. This scope is the default value if no other scope is specified.
2. **prototype**: Scopes a single bean definition to any number of object instances.
   A bean with the prototype scope will return a different instance every time it is requested from the container. The client code must clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding. To get the Spring container to release resources held by prototype-scoped beans, try using a custom bean post-processor, which holds a reference to beans that need to be cleaned up.
3. **request**: Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring **ApplicationContext**.
4. **session**: Scopes a single bean definition to the lifecycle of an HTTP **Session**. Only valid in the context of a web-aware Spring **ApplicationContext**.
5. **application**: Scopes a single bean definition to the lifecycle of a **ServletContext**. Only valid in the context of a web-aware Spring **ApplicationContext**.
6. **websocket**: Scopes a single bean definition to the lifecycle of a **WebSocket**. Only valid in the context of a web-aware Spring **ApplicationContext**.

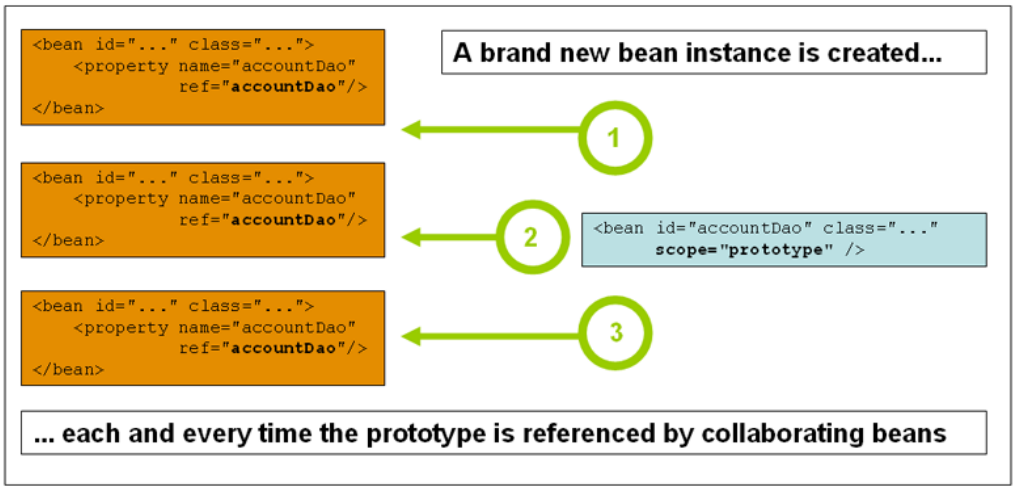
Singleton Scoped Bean Example

```
1.  @Bean
2.  @Scope("singleton")
3.  //or
4.  //@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
5.  public Laptop myLaptop() {
6.      return new Laptop();
7.  }
```



Prototype Scoped Bean Example

```
1.  @Bean
2.  @Scope("prototype")
3.  //or
4.  //@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
5.  public Laptop myLaptop() {
6.      return new Laptop();
7.  }
```



Note: Additional **Custom Scopes** can also be created and configured using a **CustomScopeConfigurer**. An example would be the **flow scope** added by Spring Webflow. In most cases, you may only deal with the Spring's core scope i.e., **singleton** and **prototype**.

Q7. What are Advantages & Disadvantages of DI?

Answer: The process of injecting and converting coupled or dependent objects into decoupled or independent objects is called Dependency Injection. It is a process that allows concurrent and or independent development. Through dependency injection, two developers can independently develop

classes that use each other, while only needing to know the interface the classes will communicate through. Moreover, dependency injection decreases coupling between classes and its dependency.

Advantages of Dependency Injection:

1. Dependency injection allows a client the flexibility of being configurable.
2. Classes are more modular, as they depend only on the interface of passed-in dependencies. Class behavior can be changed by swapping out a new component.
3. Centralized configuration can be used to externalize a system's configuration details into configuration files, allowing the system to be reconfigured without recompilation.
4. Separate configurations can be written for different situations that require different implementations of components. This includes, but is not limited to, testing.
5. Due to decoupling, the reusability of the code is increased.
6. Improved code maintainability.
7. It can be applied to legacy code as a code refactoring.
8. The ease of testing is often the first benefit noticed when using dependency injection.
9. It allows reduction of boilerplate code in the application objects, since all work to initialize or set up dependencies is handled by a provider component.
10. Improves application testing, since stubs can be substituted for any dependency.

Even though there are uncountable benefits of using Dependency Injection, it is of utmost importance to check the drawbacks and disadvantages provided by it. From encouraging dependence on a dependency injection framework to forcing complexity to move out of classes and into the linkages between classes, which might not always be desirable or easily managed, there are several disadvantages of dependency injection.

Disadvantages of dependency Injection:

1. Using many instances together can become a very difficult if there are too many instances and many dependencies that need to be resolved.
2. Code can become harder to understand.
3. Dependency injection can make code difficult to trace (read) because it separates behavior from construction. This means developers must refer to more files to follow how a system performs.

Q8. Why is Dependency Injection Useful?

Answer: DI a very useful technique for testing, since it allows dependencies to be mocked or stubbed as required. Every layer (DAO, service, controller) of application can be tested independently.

Explanation: Dependency injection plays an important role in test driven development of a software, and it is a design paradigms that is used by software programmers all over the world. Moreover, this framework takes care of creating complicated and advanced objects for developers without allowing them to worry about providing right elements or ingredients. However, the importance of dependency injection is extremely high for unit testing, validation and exception management, as it is tremendously helpful to them in various ways. Hence, elaborated here is the use of dependency injection in unit testing and validation or exception management.

Unit Testing: In unit testing, dependency injection enables one to replace complex dependencies, such as databases, with mocked implementations of those dependencies. This further allows them to completely isolate the code that is being tested by the team of expert testers.
Validation or Exception Management: With the assistance of dependency injection, an individual can inject additional code between the available dependencies. Also, it is possible to inject exception management logic or validation logic, which means that the developer no longer needs to write similar logic for every phase of development.

Spring JDBC

Q1. What is Spring JDBC?

Answer: Spring JDBC is not separate project, but it is part of core Spring framework's data access feature, usually referred to as Spring JDBC module. It takes care of all low level common JDBC operation and allows developer to focus only on business logic.

Explanation: Spring JDBC provides all of the Spring abstractions over the plain JDBC **DataSource** that you can use with the Spring Framework. The **JdbcTemplate** is part of this module, which allows you to execute SQL statements and extract objects from **ResultSet** s without dealing with exception handling or the nasty details of properly closing statements, connections and the like. Spring provides @Transactional annotation which can be applied on the class or on any of the methods, which allows the framework to inject transactional logic before and after the running method, mainly for starting and committing the transaction.

Q2. What configurations does Spring JDBC & developer need to perform in application?

Answer: The table shows which actions Spring takes care of and which actions are your responsibility.

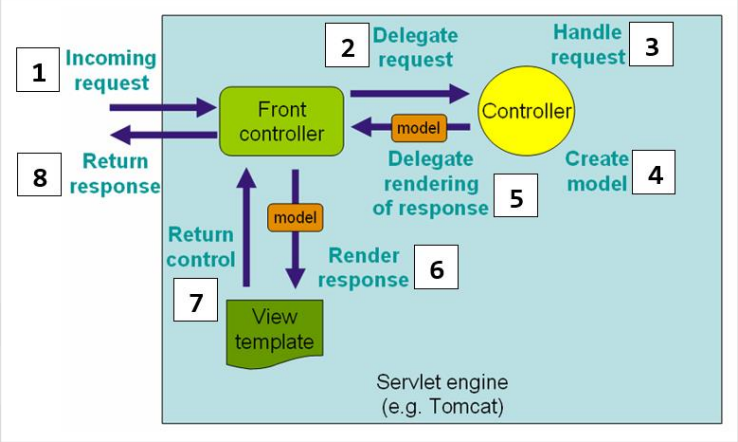
Steps	Action	Spring	Developer
1	Define connection parameters.		X
2	Open the connection.	X	
3	Specify the SQL statement.		X
4	Declare parameters and provide parameter values		X
5	Prepare and run the statement.	X	
6	Set up the loop to iterate through the results (if any).	X	
7	Do the work for each iteration.		X
8	Process any exception.	X	
9	Handle transactions.	X	
10	Close the connection, the statement, and the resultset.	X	

Spring MVC

Q1. Describe the architecture of Spring MVC.

Answer: Spring MVC is the web component of Spring's framework which provides a rich functionality for building robust Web Applications. The Spring MVC Framework is architected and designed in such a way that every piece of logic and functionality is highly configurable. It can integrate effortlessly with other popular Web Frameworks like Struts, WebWork, Java Server Faces etc. Spring is not tightly coupled with Servlets or JSP to render the View to the Clients it can be integrated with other view technologies like Velocity, Freemarker etc.

Explanation: Spring MVC, as many other web frameworks, is designed around the front controller pattern where a central **Servlet**, the **DispatcherServlet**, provides a shared algorithm for request processing, while actual work is performed by configurable delegate components. The **DispatcherServlet**, as any Servlet, needs to be declared and mapped according to the Servlet specification by using Java configuration or in web.xml. In turn, the **DispatcherServlet** uses Spring configuration to discover the delegate components it needs for request mapping, view resolution, exception handling, and more.



The request processing workflow in Spring Web MVC (high level):

- The **DispatcherServlet**, as any **Servlet**, needs to be declared and mapped according to the Servlet specification by using Java configuration or in **web.xml**. In turn,

```
<web-app>
    <servlet>
        <servlet-name>app</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>app</servlet-name>
        <url-pattern>/app/*</url-pattern>
    </servlet-mapping>
</web-app>
```

the **DispatcherServlet** uses Spring configuration to discover the delegate components it needs for request mapping, view resolution, exception handling etc.

- **DispatcherServlet** expects a **WebApplicationContext** (an extension of a plain **ApplicationContext**) for its own configuration. **WebApplicationContext** has a link to the **ServletContext** and the **Servlet** with which it is associated.
- The **DispatcherServlet** delegates to special beans to process requests and render the appropriate responses. By "special beans" we mean Spring-managed Object instances that implement framework contracts. Few special beans detected by the DispatcherServlet are:
    a. **HandlerMapping** - Map a request to a defined specific handler (e.g. @RequestMapping annotated methods).
    b. **ViewResolver** - Resolve logical String-based view names returned from a handler to an actual View with which to render to the response. Configuring view resolution is as simple as adding viewResolver beans to your Spring configuration. The Spring Framework has a built-in integration for using Spring MVC with JSP and JSTL. When developing with JSPs, you typically declare an **InternalResourceViewResolver** bean. **InternalResourceViewResolver** can be used for dispatching to any Servlet resource but in particular for JSPs. As a best practice, Spring strongly encourages placing all your JSP files in a directory under the **'WEB-INF'** directory so there can be no direct access by clients.

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

Spring Boot

Q1. What is Spring Boot?

Answer: Spring Boot is basically an extension of the Spring framework, it allows us to build a stand-alone application based on the jar dependencies you have enabled, with less or nearly zero configuration.

Explanation: It takes an opinionated 'starter' dependencies to simplify the build and application configuration, which paves the way for a faster and more efficient development ecosystem. Mostly with Spring Boot, we just need to notify Spring how many plugins we want to use, and we'll get a quick setup for them. We can use the spring initializer to pull our application into all the dependencies we need, thereby reducing our development time. Spring Boot assists in faster development as it has a lot of starter projects that help you get moving very quickly. Many non-functional features are also included, such as embedded servers, some metrics, health checks, safety, etc. In short, with minimal invasion of code, it makes Spring-based application creation simpler. It comes with the starter definition in the

pom.xml file that takes care of installing the JAR dependencies depending on the Spring Boot Requirement. All in all, Spring Boot is a project initializer based on Spring. It prevents you from writing long code with features such as auto-configuration and lets you avoid excessive configuration.

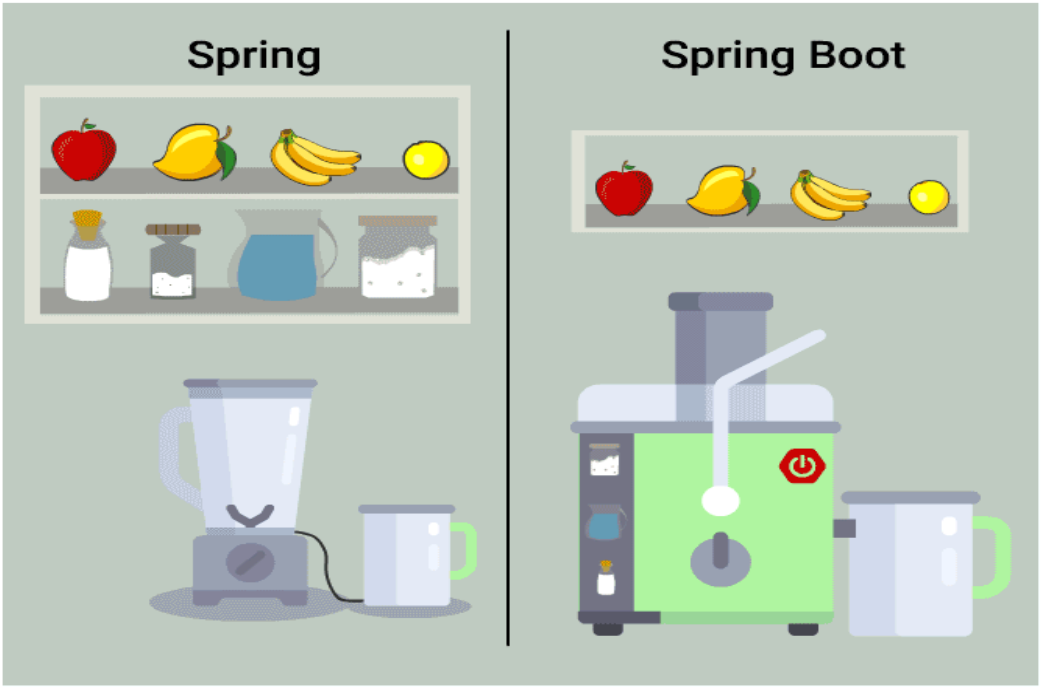
Q2. When should you use Spring Boot vs. Spring?

Answer: It is better to use Spring Boot if we want to develop a simple Spring-based application or RESTful services.

Q3. What is difference between Spring Boot vs. Spring?

Answer:

Spring Boot	Spring
Autoconfiguration is the primary feature of Spring Boot	Dependency Injection is the primary feature of Spring
Spring Boot Framework is widely used to develop REST APIs.	Spring Framework is a widely used for building Java SE, Java EE applications.
It aims to shorten the code length and provide the easiest way to develop Web Applications.	It aims to simplify Java EE development that makes developers more productive.
The primary feature of Spring Boot is Autoconfiguration. It automatically configures the classes based on the requirement.	The primary feature of the Spring Framework is dependency injection.
It helps to create a stand-alone application with less configuration.	It helps to make things simpler by allowing us to develop loosely coupled applications.
It reduces boilerplate code.	For smaller tasks, developers need to write a boilerplate code
Spring Boot offers embedded server such as Jetty and Tomcat, etc.	To test the Spring MVC project, we need to set up the sever explicitly.
Spring Boot comes with the concept of starter in pom.xml file that internally takes care of downloading the dependencies JARs based on Spring Boot Requirement.	Developers manually define dependencies for the Spring project in pom.xml.
There is no requirement for a deployment descriptor i.e., web.xml file.	For Spring MVC applications a deployment descriptor is required.



Q4. What are some of the annotations you've used in Spring Boot?

Answer: Spring Annotations are a form of metadata that provides data about a program. Annotations are used to provide supplemental information about a program. Spring annotations present in the **org.springframework.boot.autoconfigure** and **org.springframework.boot.autoconfigure.condition** packages are commonly known as Spring Boot annotations. Some of the annotations that are available in this category are as follows:

1. **@SpringBootApplication** - This annotation is used to mark the main class of a Spring Boot application. It encapsulates @SpringBootConfiguration, @EnableAutoConfiguration, and @ComponentScan annotations with their default attributes.
2. **@ComponentScan** - It is used when we want to scan a package for beans. Generally, @ComponentScan annotation is used with @Configuration annotation to specify the package for Spring to scan for components. We can also specify the base packages to scan for Spring Components. It tells Spring in which packages you have annotated classes that should be managed by Spring. For example, if you have a class annotated with @Controller which is in a package that is not scanned by Spring, you will not be able to use it as a Spring controller. So, we can say @ComponentScan enables Spring to scan for things like configurations, controllers, services, and other components that are defined.
3. **@EnableAutoConfiguration** - It auto-configures the bean that is present in the class path and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e., @SpringBootApplication. For example, when we illustrate the spring-boot-starter-web dependency in the class path, Spring boot auto-configures Tomcat, and Spring MVC. The package of the class

declaring the @EnableAutoConfiguration annotation is considered as the default. Therefore, we need to apply the @EnableAutoConfiguration annotation in the root package so that every sub-packages and class can be examined.

4. **@Configuration** - It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.
5. **@SpringBootConfiguration** - It can be used as an alternative to Spring's standard @Configuration annotation so that configuration can be found automatically. Most Spring Boot Applications use @SpringBootConfiguration via @SpringBootApplication.

Q5. How to create Spring Boot project? Or What is Spring Initializr? How would I go about writing a spring boot application to expose endpoints?

Answer: The easiest way to build Spring Boot project is to used Spring Initializr module. Spring Initializr provides an extensible API to generate JVM-based projects, and to inspect the metadata used to generate projects, for instance to list the available dependencies and versions.

Explanation: With Spring Initializr, there is minimal code involved and the service has a very rich configuration structure, allowing you to define not only the values of various project attributes but also the list of dependencies and the constraints to apply to them. The API can be used standalone or embedded in other tools (e.g., it is used in major IDEs such as Spring Tool Suite, IntelliJ IDEA Ultimate, NetBeans and VSCode).

Steps:

1. Navigate to **https://start.spring.io**. This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Gradle or **Maven** and the language you want to use. This guide assumes that you chose Java.
3. Click **Dependencies** and select **Spring Web, Spring Data JPA,** and **MySQL** Driver.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.
6. Open the project into any java IDE.
7. Configure standard package names.
8. Define various configuration details for your spring application e.g., DB server details under each **application-<spring-profile-name-by-region>.properties** file.
9. Configure various entities used in project using respective ORM provider.
10. Implement DAO layer repositories using either **JpaRepository** or **CrudRepository** (We prefer using **JpaRepository** due to its additional benefits) and using **@Repository** annotation.
11. Implement service layers using more business specific functionalities by injection one or more repositories.
12. Implement controller layer using **@Controller** or **@RestController** depending upon business requirements.

13. Expose various endpoint methods using appropriate request mapping annotations like. **@GetMapping, @PostMapping, @PutMapping** or **@DeleteMapping**
14. Test Application using any client like SoapUI, Postman etc.

Q6. What are the standard package names which are followed by any enterprise java application?

Answer: Usually there is no restriction on whatever the package name you provide for your application, but as per industry standards and enterprise application being divided into various layered structure the standard package structures which can be defined in project are listed below -

1. **com.revature.model** - This will have all your POJOs and JPA entities defined.
2. **com.revature.repositories** or **com.revature.dao** - This will have all you @Repositries or any DAO interface & classes defined.
3. **com.revature.service** - This will have all you @Service interface & classes defined.
4. **com.revature.controller** - This will have all you @Controller or @RestControllers classes defined.
5. **com.revature.constants** - This will have all your classes with private constructor and all public static final constants defined.
6. **com.revature.util** - This will have all your own utility classes defined e.g.pdf, csv export file utilities, encryption/decryption.
7. **com.revature.config** - This will have all your spring @Configuration classes with @Bean definitions.
8. **com.revature.dto** - This will have all your Data Transfer Object (DTO) classes defined which are usually subsets of attributes from different POJO and Entity classes.
9. **com.revature.test.unit** - This will have all your application jUnit test cases defined.
10. **com.revature.test.api** - This will have all your application integration test cases defined.

Q7. What is the standard Spring profile per region properties file names followed by any enterprise java application?

Answer: Usually there is no restriction on whatever the profile names you use but as per industry standards you can use below spring profile and respective properties file name-

1. **application.properties** - Spring Default profile
2. **application-dev.properties** - Spring Dev Profile for Development Region
3. **application-int.properties** - Spring Dev Profile for Integration Testing Region
4. **application-uat.properties** - Spring Dev Profile for User Acceptance Testing Region
5. **application-staging.properties** - Spring Dev Profile for User Staging/Performance Testing Region

6. `application-preprod.properties` - Spring Dev Profile for Pre-Production or Sandbox Region
7. `application-prod.properties` - Spring Dev Profile for PROD Region

Q8. Which IDE can be used to build Spring Boot application?

Answer: You can use any latest version of Eclipse, Spring Tool Suite, IntelliJ Idea, Visual Studio Code or any other IDE compliant with your enterprise guidelines. Out of all the IDE spring recommended to use Spring Tool Suite since it has Spring Initializr integration integrations preconfigured for other IDEs you may need to configure Spring plugins manually.

Q9. How does Spring Boot Application Bootstraps?

Answer: The entry point of execution of a Spring Boot application is the class that is annotated with @SpringBootApplication.

```
1.  @SpringBootApplication
2.  public class Application {
3.      public static void main(String[] args) {
4.          SpringApplication.run(Application.class, args);
5.      }
6.  }
```

Spring boot application uses embedded container to run the application. Spring Boot uses the public static void main entry point to launch an embedded web server. The binding of the Servlet, Filter, and ServletContextInitializer beans from the application context to the embedded servlet container is take care internally. It automatically scans all the classes in the same package or sub packages of the Main-class for components. In addition, Spring Boot provides the option of deploying it as a web archive in an external container for this we have to extend the SpringBootServletInitializer as follows, Here the external servlet container looks for the Main-class defined in the META-INF file of the web archive, and the SpringBootServletInitializer will take care of binding the Servlet, Filter, and ServletContextInitializer.

```
1.  @SpringBootApplication
2.  public class Application extends SpringBootServletInitializer {
3.      // ...
4.  }
```

Spring Data, Spring Data JDBC & Spring Data JPA

Q1. What is Spring Data?

Answer: The goal of Spring Data is to make it easier to work with different kinds of databases, from traditional relational databases to NoSQL databases. Spring Data is the project which consists of multiple modules (sub-projects). Below are the three main modules widely used from Spring Data project-

1. Spring Data Commons – It provides shared infrastructure across the Spring Data projects. It contains technology neutral repository interfaces as well as a metadata model for persisting Java classes.

2. Spring Data JDBC - Spring Data repository support for JDBC. It does NOT offer caching, lazy loading, write behind or many other features of JPA. This makes Spring Data JDBC a simple, limited, opinionated ORM.
3. Spring Data JPA - Spring Data repository support for JPA.

Explanation: Spring Data and the sub projects are built on top of the Spring Framework. Spring Data's mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store. It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services. This is an umbrella project which contains many subprojects that are specific to a given database.

Q2. What is Spring Data JDBC?

Answer: Spring Data JDBC is the project which gives benefits of Spring Data, combined with those of JDBC-

1. It provided repositories with CRUD methods out of the box i.e. it provides CrudRepository interface for generic CRUD operations on a repository for a specific type.
2. Support for @Query annotations.
3. Nice and easy ways to extend repositories with query methods (we can just define our interface to have e.g. findByLastName method and Spring generates it for on the fly).
4. Support for PagingAndSortingRepository interface which is extension of CrudRepository that provides additional methods to retrieve entities using the pagination and sorting.
5. Nice integration in the Spring infrastructure, for transaction handling, dependency injection, error translation.

Q3. What is Spring Data JPA?

Answer: Spring Data JPA provides repository support for the Java Persistence API (JPA). It eases development of applications that need to access JPA data sources. Just like JPA, Spring Data JPA cannot work without a JPA provider. Spring Data JPA can work with Hibernate, Eclipse Link, or any other JPA provider. Spring Data JPA does a lot of things over Spring Data JDBC

1. Caching (1st, 2nd level, and query cache)
2. Automated creation of instances from queries
3. Navigation between entities
4. Lazy loading
5. Dirty checking / tracking of changes to entities

Q4. What is Spring Data JPA vs Spring Data JDBC?

Answer: The key differences are listed below-

#	Spring Data JPA	Spring Data JDBC

1	Database independent and portable	Generally, Database Specific
2	Introduces complexity through Object-Relational Mapping	Less complex, than Spring Data JPA
3	Automatic DB schema generation based on the entities	Manual DB schema generation through DDL commands
5	Query annotated with JPQL code and native SQL	Query annotated with only native SQL
7	Models relationships between entities through annotations as @OneToMany, @Embedded, etc.	No relationship modeling annotations.
8	Caching and lazy loading	No caching, no lazy loading
9	Sessions and dirty tracking	No sessions, no dirty tracking

Q5. What is JPA?

Answer: The Java Persistence API, sometimes referred to as JPA, is a Java framework managing relational data in applications using the Java Platform, Standard Edition (JavaSE) and Java Platform, Enterprise Edition(JavaEE).

Q6. How to use Spring Data JPA in Spring Boot Project?

Answer: There are below steps you can follow in order to incorporate Spring Data JPA into Spring Boot project-

1. First you need to set up connection with your desired database in spring application using properties like url, driver, username, password etc.
2. Add required dependencies in project, e.g. for maven below JPA dependency need to be added into pom.xml-

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
	<groupId>mysql</groupId>
	<artifactId>mysql-connector-java</artifactId>
	<scope>runtime</scope>
</dependency>
```

3. Define the required ORM entities (model classes) with applicable annotations.
4. Next you need to create a repository interface and extend it to the **JpaRepository** interface
5. The **JpaRepository** will provide you set of default methods to perform basic CRUD operations, I addition to that if you need some add custom, complex queries you can define those using standard provided by Spring Data JPA.
6. Next you need to inject the created repository interface in applicable service layers defined in application.

7. The service layer is further injected into Controller layer to achieve the desired CRUD functionality.

Spring Data JPA Example

```
@Entity
@Table(name = "employees")
@Data
@AllArgsConstructor
@NoArgsConstructor
@EntityListeners(LastUpdateListener.class)
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @NotBlank
    @Column(name = "first_name", nullable = false)
    private String firstName;

    @NotBlank
    @Column(name = "last_name", nullable = false)
    private String lastName;

    @NotBlank
    @Email
    @Column(name = "email_address", nullable = false)
    private String emailId;

    @Min(18)
    @Max(65)
    private Integer age;

    @Pattern(regexp = "^[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}$")
    private String ipAddress;
}

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long>{
    //.. other custom queries
}

@Service
public class EmployeeServiceImpl implements EmployeeService{
    @Autowired
    private EmployeeRepository employeeRepository;
    //.. other service layer code
}

@RestController
@CrossOrigin(origins = "http://localhost:8081")
@RequestMapping("/api/v1")
public class EmployeeController {
    @Autowired
    private EmployeeService employeeService;
    //.... other controller layer code
}

```

Appendix

- Spring Web MVC: https://docs.spring.io/spring-framework/docs/current/reference/html/web.html
- Spring IoC: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#spring-core
- Dependency Injection: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-collaborators
- @Autowired : https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-autowired-annotation
- autowire mode: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-autowire
- Spring Data JDBC, References, and Aggregates: https://spring.io/blog/2018/09/24/spring-data-jdbc-references-and-aggregates