In the context of the Spring Framework, “wiring up the plumbing code” refers to the process of setting up connections between different parts of your application, such as between classes and their dependencies. This is often referred to as “dependency injection”.

Spring provides a comprehensive programming and configuration model for modern Java-based enterprise applications1. A key element of Spring is infrastructural support at the application level: Spring focuses on the “plumbing” of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments.

Spring can automatically inject dependencies. The Spring container detects those dependencies specified in the configuration file and establishes the relationship between the beans. This is referred to as autowiring in Spring.

For example, if you have a class ArbitraryClass that depends on another class or resource, you can use annotations like @Autowired, @Resource, or @Inject to tell Spring to automatically provide an instance of the needed class or resource when an instance of ArbitraryClass is created3. This way, you don’t have to manually create and manage these dependencies in your code, which makes it cleaner and easier to maintain.

Here’s a simple example:

@Autowired

ArbitraryClass arbObject;

Copiar

In this case, Spring will automatically create an instance of ArbitraryClass and assign it to arbObject when needed3. This is much simpler and more manageable than having to manually instantiate arbObject with something like ArbitraryClass arbObject = new ArbitraryClass();.

So, when we say that Spring “wires up the plumbing code”, we mean that it handles these kinds of low-level details for you, allowing you to focus on the higher-level business logic of your application. It’s one of the reasons why Spring is such a popular framework for Java development.

Question 1

In an XML file how is bean injection achieved?

Bean injection in an XML file is achieved in Spring through Dependency Injection. This can be done using properties, constructors, or static factories. For instance, you can wire dependencies together using an XML-based configuration.

Correct

Through the use of a <property> tag within the bean definition and a “ref” attribute to the id or name of the desired bean. An object instance is created and uses the target properties setter to inject it into the enclosing bean.

Question 2

What is Simple value Injection?

Simple value Injection refers to the process of injecting simple values like strings, integers, etc., into beans. This is typically done using the @Value annotation in Spring.

Correct

The actual value in the “value” attribute of an xml <property> tag uses reflection to see what the target property is, pasrses the String value form the XML file through the desired PropertyEditor and injects it through the target properties setter.

Question 3

How is Spring achieving inversion of Control?

Spring achieves Inversion of Control (IoC) through Dependency Injection (DI). IoC is a design principle where the control flow of the program is inverted, i.e., instead of the application code controlling the flow, the external framework does. This is achieved in Spring by creating and managing beans (objects) and their dependencies.

Correct

Encapsulates Construction and Bean Dependencies through configuration that describes the “rules” that govern what dependency is injected into what bean. Because our code uses Interface types for these dependencies, actual implementation classes are hidden from the code dependency.

Question 1

What is the javax equivalent of Springs @Autowired and @Qualifier annotations?

The javax equivalent of Spring’s @Autowired and @Qualifier annotations is the @Inject annotation. However, it’s important to note that @Inject does not have an exact equivalent to @Qualifier. For finer control over dependency injection in javax, you might need to create custom annotations or use @Named.

Correct

@Inject and @Named

Question 2

What is the @PostConstruct annotation used for?

The @PostConstruct annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization. This method must be invoked before the class is put into service. It’s part of the JSR-250 specification, which is widely supported in Java frameworks.

Correct

Initialization through programmatic logic once Constructor and Setter Injectin has completed, essentially like java’s instance Initializer but managed by Spring

Question 3

What annotation would you use for simple value injection?

For simple value injection in Spring, you would use the @Value annotation. This annotation can be used for injecting values into fields in Spring-managed beans, and it can be applied at the field or constructor/method parameter level.

Correct

@Value

Question 1

Why use Java Configuration classes, when is it verified?

Java Configuration classes in Spring Framework are used to write most of your Spring configuration without XML but with the help of few Java-based annotations. These classes are verified when the application context is created and the beans are initialized.

Correct

It provides a type-safe configuration, checked at compile time rather than run time such as XML. In the Bean factory methods, you have the power of Java to create the bean in any fashion that you wish. The configuration is external rather than inline like Annotational Configurations.

Question 2

What key annotations are used in Java Configuration classes?

The key annotations used in Java Configuration classes are @Configuration, @Bean, @Import, and @ComponentScan. The @Configuration annotation is used on classes which define beans. The @Bean annotation is used at the method level. The @Import annotation allows for loading @Bean definitions from another configuration class. The @ComponentScan annotation is used with the @Configuration annotation to specify the packages to scan for annotated components.

Correct

@Configuration and @Bean, in fact you can inject beans into Configuration classes so @Value, @Autowired and @Qualifier, @Inject and @Named are also commonly used

Question 3

Using JUnit 5 SpringExtension, how does the Test class access your Spring Managed Beans?

In JUnit, the SpringExtension integrates the Spring TestContext Framework into JUnit’s Jupiter programming model. Test classes can access Spring Managed Beans by using the @Autowired annotation to inject beans into class variables, or by using the @ContextConfiguration annotation on the test class, which creates its own ApplicationContext for the test class that includes the test class itself.

Correct

Using the @ContextConfiguration annotation on the test class, the annotation points to an XML file or Java configuration classes and creates its own ApplicationContext for the test class that includes the test class itself. Subsequently, Spring Managed beans can be injected into it directly.

Question 1

How do we introduce EL expressions into a @Value annotation?

In Spring, you can introduce EL expressions into a @Value annotation by using the ${...} syntax. For example, if you have a property named myProperty in a properties file, you can inject its value into a field like this: @Value("${myProperty}") private String myField;

Correct

The syntax is “${...}”, the enclosed expression resolves to picking up a literal match from the Spring Environment Object that has, perhaps, been read from a properties file via a PropertySourcesPlaceholderConfigurer bean.

Question 2

What makes SPEL more powerful than EL?

Spring Expression Language (SpEL) is more powerful than Unified EL because it supports additional features such as method invocation and basic string templating functionality. It also supports querying and manipulating an object graph at runtime.

Correct

EL only reads literals for textural replacement, while SPEL has the ability to access Spring Managed beans, execute methods on them, filter Collections and access System Properties.

Question 3

How do you filer a Collection using SPEL?

In SpEL, you can filter a collection using the .?[...] syntax. For example, if you have a list of integers and you want to filter out the even numbers, you can do something like this: @Value("#{list.?[#this%2==0]}") private List<Integer> evens;

Correct

The syntax is “#{id.property.?[test]}”. Using the .? notation on a Collection property means that we want to gather all matches of the proceeding test.

Question 1

What do we use Conditional beans for?

Conditional beans in Spring are used to control whether a bean is included in the application context based on certain conditions. This is useful in scenarios where certain beans may not work in specific environments, such as test environments.

Correct

Conditionals are used to omit or include beans in the ApplicationContext based off other bean dependencies, environment in a single re-useable context configuration across development environments. For example, a different Service bean could be used in “test” than in “prod”

Question 2

What are some common provided Conditional Annotations from Spring Boot?

Spring Boot provides several predefined conditional annotations, including @ConditionalOnProperty, @ConditionalOnExpression, @ConditionalOnBean, @ConditionalOnMissingBean, @ConditionalOnResource, @ConditionalOnClass, @ConditionalOnWebApplication, @ConditionalOnJava, @ConditionalOnCloudPlatform, @ConditionalOnWarDeployment, @ConditionalOnJndi, @ConditionalOnSingleCandidate, @ConditionalOnManagementPort, and @ConditionalOnAvailableEndpoint

Correct

@ConditonOnBean, @ConditionOnExoression, @ConditionOnProperty, @ConditionOnResource

Question 3

What is the key Interface to Implement to create a Custom Condition?

To create a custom condition in Spring, you need to implement the Condition interface.

Correct

org.springframework.context.annotation.Condition and implement it matches method