DUMPSBOSS.

Core-Spring (based on Spring 3.2)

SpringSource CoreSpringV3.2

Version Demo

Total Demo Questions: 11

Total Premium Questions: 117

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

Topic Break Down

Topic	No. of Questions
Topic 1, The Spring Container	62
Topic 2, Aspect-Oriented Programming	19
Topic 3, Data Access	10
Topic 4, Transaction Management	15
Topic 5, Spring MVC	6
Topic 6, Mix Questions	5
Total	117

QUESTION NO: 1

Identify the correct statement(s) about the following pointcut expression:

execution(* rewards..restaurant.*.*(*)) (select one or several answers)

A. There may be several directories between 'rewards' and 'restaurant'

B. There is no restriction on the class name

C. The target method may have several arguments

D. The return type must not be "void"

ANSWER: A B

Explanation:

The expression `execution(* rewards..restaurant.*.*(*))` uses Spring AOP's execution pointcut syntax to select method-execution join points. The `rewards..restaurant` portion is a type-pattern package expression. In Spring's AspectJ-style pointcut language, `..` represents zero or more package segments, so types can be located in `rewards.restaurant` itself or in a package path containing intervening package levels, such as `rewards.loyalty.restaurant`. Thus, there may be several package directories between `rewards` and `restaurant`.

The first `*` after `restaurant.` is the class-name portion of the type pattern. It matches any class name in the selected package location, so the expression imposes no restriction on the class name. The initial `*` similarly permits any method return type. Spring documents the `execution` designator and its method-signature pattern elements, including wildcard type patterns and the `..` package wildcard, in its AOP reference documentation. See Spring Framework 3.2 AOP reference and AspectJExpressionPointcut Javadoc.

QUESTION NO: 2

Identify the correct statement(s) regarding the tag (Select one or multiple answers)

A. Can only be used with AspectJ (as opposed to Spring AOP)

B. Enables the detection of @Aspect annotated classes

C. Allows to define a list of names to include so Spring is not going to scan all the beans at startup

ANSWER: B C

Explanation:

The `<aop:aspectj-autoproxy/>` element activates Spring's support for the `@Aspect` annotation style. Once it is enabled, Spring's auto-proxy creator locates beans defined as aspects, recognizes their pointcut and advice annotations, and applies the resulting Spring AOP advice to eligible target beans. The aspect class itself must still be registered as a Spring bean, for example through component scanning or an explicit bean definition; the element enables the infrastructure that detects and uses that `@Aspect` bean.

The element can also contain one or more `<aop:include name="..."/>` child elements. These specify bean-name patterns that limit which aspect beans are eligible for automatic proxy creation. This is useful when an application context has many beans and only selected named beans should be considered as `@Aspect` candidates, reducing the set examined by the auto-proxy infrastructure at startup. The configuration uses Spring AOP proxying while adopting AspectJ's annotation-based programming model. Spring documents both annotation-style aspect support and the include-pattern facility in its AOP schema documentation: Spring Framework 3.2 AOP reference and Spring AOP schema reference.

QUESTION NO: 3

Which of the following statements are true regarding initialization callbacks for a Spring bean? (Select one or several answers)

A. Dependency injection occurs before initialization callbacks are invoked

B. afterPropertiesSet() is invoked before a configured custom init-method

C. A @PostConstruct method is invoked after afterPropertiesSet()

D. A destruction callback is invoked as part of bean initialization

ANSWER: A B

Explanation:

Spring performs dependency injection before initialization callbacks. Where all three mechanisms are configured, a method annotated with `@PostConstruct` is invoked during pre-initialization processing, followed by `InitializingBean.afterPropertiesSet()`, followed by a custom method configured through the bean definition's `init-method` attribute. This ordering allows each initialization callback to use already injected dependencies.

The interfaces `InitializingBean` and `DisposableBean` are Spring-specific lifecycle contracts. A destruction callback is not invoked during initialization; it is invoked when the container destroys an eligible bean, such as when a configurable application context is closed. See the Spring Framework 3.2 bean lifecycle documentation.

QUESTION NO: 4

Which of the following statements is NOT true concerning Setter Injection or Constructor Injection? (Select one)

A. Constructor injection is useful when you must have an instance of a dependency class before your component is used

B. Setter injection is useful if a component can provide its own defaults

C. Using the @Autowired annotation, setter injection also works when the setter method is private

D. Using setters promotes immutability

ANSWER: D

Explanation:

Using setters promotes immutability is the statement that is not true. Immutability means that an object's state cannot be changed after the object has been constructed. Setter injection supplies dependencies by invoking methods after object construction, so it inherently allows a dependency reference to be changed later through another setter invocation. This is useful for optional dependencies and for components that can operate with a sensible default configuration, but it does not make the resulting object immutable.

Constructor injection is the Spring-recommended approach for required dependencies partly because constructor parameters can be assigned to `final` fields. Once construction has completed, those fields cannot be reassigned, making the dependency relationship stable for the lifetime of the bean and supporting an immutable design. Spring's reference documentation specifically distinguishes constructor-based dependency injection for mandatory dependencies from setter-based injection for optional dependencies. The Spring Framework team also recommends constructor injection for required dependencies and notes that it enables application components to be implemented as immutable objects. See the Spring 3.2 reference documentation on constructor-based injection and the current Spring documentation on dependencies and configuration.

QUESTION NO: 5

Which of the following methods is NOT provided by the JmsTemplate? (select one)

A. setDefaultDestination

B. onMessage (asynchronous call)

C. convertAndSend

D. receiveAndConvert (synchronous call)

ANSWER: B

Explanation:

`onMessage (asynchronous call)` is correct because it is not a method on Spring's `JmsTemplate`. In JMS, asynchronous consumption is callback-based: a message listener receives a delivered JMS `Message` through the `onMessage(Message)` callback defined by the JMS `MessageListener` contract. Spring applications commonly use a message-listener container, such as `DefaultMessageListenerContainer`, to create and manage the JMS consumers and invoke that callback on an application listener. This container-based arrangement supplies the asynchronous listener lifecycle, threading, transaction participation, recovery, and resource management that are needed for reliable background message consumption.

`JmsTemplate` instead serves as Spring's synchronous, template-style abstraction for JMS operations. Its API is designed for executing JMS operations with Spring-managed resource handling, while asynchronous reception is delegated to listener infrastructure and a listener implementation. Therefore, the callback name `onMessage` identifies the message-listener programming model rather than an operation offered directly by `JmsTemplate`. See the Spring 3.2 JmsTemplate API and the Spring reference documentation on JMS message listeners.

QUESTION NO: 6

Which is the correct statement regarding the relationship between a DispatcherServlet ApplicationContext and a root ApplicationContext (select one)

A. The root ApplicationContext can reference beans in the DispatcherServlet Application context, but not vice-versa

B. The DispatcherServlet ApplicationContext can reference beans in the root Application context, but not vice-versa

C. The DispatcherServlet ApplicationContext and the root ApplicationContext can reference each other's beans

D. The DispatcherServlet ApplicationContext and root ApplicationContext are completely isolated and cannot reference each other's beans

ANSWER: B

Explanation:

The DispatcherServlet ApplicationContext can reference beans in the root Application context, but not vice-versa. In a traditional Spring MVC web application, the root context is created by the context loader infrastructure and is used for shared, typically non-web concerns such as services, repositories, data sources, transaction management, and security. Each DispatcherServlet then creates its own child WebApplicationContext for MVC-specific infrastructure and web-layer beans, such as controllers, handler mappings, view resolvers, and related configuration.

Spring ApplicationContexts support a parent-child hierarchy. Bean lookup in a child context includes its parent, so a controller defined in the DispatcherServlet context can inject or otherwise obtain a service defined in the root context. This arrangement lets multiple DispatcherServlet instances share common application services while retaining separate servlet-specific web configurations. The parent context does not perform reverse lookup into child contexts; therefore, root-level beans should not depend on servlet-specific beans. Spring's MVC documentation explicitly describes the root context as the parent of servlet-specific contexts and explains that servlet contexts inherit beans from that parent. See the Spring Framework 3.2 MVC context hierarchy documentation and the ApplicationContext API documentation.

QUESTION NO: 7

Identify the correct statement(s) about the following pointcut expression:

execution(* rewards.restaurant.*Service.find(..))

A. The target's type should end with "Service"

B. The target method name could be "findRestaurantById"

C. The target method should have one argument only

D. The target method may have several arguments

ANSWER: A D

Explanation:

The statements *The target's type should end with "Service"* and *The target method may have several arguments* correctly describe this execution pointcut. In Spring AOP's `execution` designator, the first `*` is the return-type pattern, so it allows a method with any return type. The declaring-type pattern is `rewards.restaurant.*Service`: it selects types directly in the `rewards.restaurant` package whose names match the `*Service` pattern, meaning their names end in `Service`. The method-name pattern is `find`, which denotes precisely that method name. Finally, `(..)` is Spring AOP/AspectJ pointcut syntax for any number of parameters, including no parameters, one parameter, or multiple parameters of any types. Consequently, a matching `find` method on a qualifying service type can accept several arguments. Spring's pointcut documentation describes `execution` expressions and identifies `(..)` as matching zero or more parameters; the AspectJ programming guide provides the same method-pattern semantics. See Spring Framework 3.2 AOP reference and AspectJ pointcut semantics.

QUESTION NO: 8

Which of the following statements defines an "aspect"? (select one)

A. An encapsulation of advice combined with a pointcut.

B. A point in the execution of a program such as a method call or field assignment

C. An expression that selects one or more join points

D. Code to be executed at a join point that has been selected by a Pointcut

ANSWER: A

Explanation:

An encapsulation of advice combined with a pointcut is an aspect in Spring AOP terminology. An aspect is the modular unit used to represent a cross-cutting concern: behavior that applies across multiple parts of an application, such as transaction management, authorization, logging, or auditing. The pointcut defines the set of join points at which the cross-cutting behavior should apply, while the advice supplies the action to run at those selected points—for example, before, after, or around a method execution. Combining these elements in one aspect keeps the cross-cutting concern cohesive and separate from the business components it affects. In Spring, an aspect is commonly implemented as a regular Java class annotated with `@Aspect`, with methods annotated to declare advice and pointcut expressions, or it can be declared through XML configuration. Spring's reference documentation explicitly describes an aspect as a modularization of a concern that cuts across multiple classes, and notes that an aspect combines pointcuts and advice. See the Spring Framework 3.2 AOP reference documentation and the @Aspect API documentation.

QUESTION NO: 9

Select which statement is true with respect to constructor injection with Spring (select one)

A. Multiple parameters can be dependency injected into a constructor

B. Using XML configuration, the constructor-arg element may be omitted if the constructor requires a single parameter

C. One single bean cannot mix constructor injection with setter injection

D. All of the above

ANSWER: A

Explanation:

Multiple parameters can be dependency injected into a constructor. In Spring constructor-based dependency injection, the container resolves the arguments required by a bean class constructor and supplies them when it creates the bean instance. A constructor may declare any practical number of dependencies, such as a repository, a service collaborator, and a configuration value. In XML bean definitions, each constructor argument can be expressed with a `<constructor-arg>` element, using a bean reference, a literal value, a type, an index, or a name where needed to disambiguate overloaded constructors. Spring matches these configured values to the constructor signature and invokes that constructor to create the object. This supports making required collaborators explicit in the class API and ensures that the fully initialized dependency set is available as soon as construction completes. The same underlying capability is also used by annotation-based configuration, where Spring can select and resolve an appropriate constructor and inject all of its parameters. See the Spring Framework reference documentation on constructor-based dependency injection and the current reference documentation on dependency injection collaborators.

QUESTION NO: 10

Consider the following class, assuming annotation-based injection has been enabled:

```
public class NotificationService {
 @Resource(name = "smsGateway")
 private MessageGateway gateway;
}
```

Which statement is true? (Select one)

A. The bean named "smsGateway" is injected if it is type compatible

B. Any bean of type MessageGateway is injected regardless of its name

C. The field can only be injected if it is declared public

D. @Resource requires the target class to implement FactoryBean

ANSWER: A

Explanation:

The field is injected with the bean named `smsGateway`. The JSR-250 `@Resource` annotation performs name-based resolution when its `name` attribute is specified. Spring first resolves the requested resource name and then verifies that the resolved bean is compatible with the field type, `MessageGateway`.

This differs from `@Autowired`, whose default resolution is primarily by type. Support for `@Resource` is supplied by `CommonAnnotationBeanPostProcessor`, which can be registered in XML using `<context:annotation-config/>`. See the Spring Framework 3.2 documentation on @Resource.

QUESTION NO: 11

Select which of the following configuration tasks would be implemented using Spring's XML "context" namespace (select one or several answers)

A. Enabling component-scanning

B. Enabling the use of the @Transactional annotation

C. Enabling the use of the @Required, @PreDestroy and @PostConstruct annotations

D. Enabling the use of the @Around annotation

ANSWER: A C

Explanation:

Enabling component-scanning is implemented with the Spring `context` namespace, specifically through `<context:component-scan>`. This element directs the container to inspect one or more base packages for stereotype-annotated classes, such as classes marked with `@Component`, `@Service`, `@Repository`, or `@Controller`, and to register the discovered classes as bean definitions. It also enables the annotation-processing infrastructure needed for commonly used dependency-injection annotations.

Enabling the use of the @Required, @PreDestroy and @PostConstruct annotations is likewise a responsibility of the `context` namespace. The `<context:annotation-config/>` element registers the relevant bean post-processors with the application context. These processors recognize `@Required` properties and lifecycle callback annotations including `@PostConstruct` and `@PreDestroy`. Consequently, this XML configuration allows Spring to validate required bean properties and invoke lifecycle methods at the appropriate points during bean initialization and destruction. Spring's reference documentation describes component scanning at Component scanning and annotation-based bean configuration at Annotation-based container configuration.