PassQueen
a.k.a dumpsqueen.com

Professional Develop VMware Spring

VMware 2V0-72.22

Version Demo

Total Demo Questions: 8

Total Premium Questions: 81

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, Spring Core	29
Topic 2, Spring Boot	21
Topic 3, Spring MVC	15
Topic 4, Spring Data	9
Topic 5, Spring Security	7
Total	81

QUESTION NO: 1

Which lifecycle annotation can be used on a Spring bean method that should run after dependency injection has completed? (Choose the best answer.)

A. @PostConstruct

B. @BeforeInjection

C. @InitializeBean

D. @AfterProperties

ANSWER: A

Explanation:

`@PostConstruct` can be placed on a no-argument instance method that should run after Spring has populated the bean's dependencies and before the bean is made available for normal use. It is commonly used for initialization that depends on injected collaborators or configuration values.

A `BeanPostProcessor` participates in the lifecycle surrounding initialization callbacks, including `@PostConstruct`. For destruction callbacks, the corresponding annotation is `@PreDestroy`. See the Spring Framework reference on @PostConstruct and @PreDestroy.

QUESTION NO: 2

Which two statements about the @Autowired annotation are true? (Choose two.)

A. @Autowired fields are injected after any config methods are invoked.

B. Multiple arguments can be injected into a single method using @Autowired.

C. By default, if a dependency cannot be satisfied with @Autowired, Spring throws a RuntimeException.

D. If @Autowired is used on a class, field injection is automatically performed for all dependencies.

E. @Autowired can be used to inject references into BeanPostProcessor and BeanFactoryPostProcessor.

ANSWER: B C

Explanation:

`@Autowired` may annotate a method with any number of parameters. When Spring creates the bean, it resolves a matching dependency for each method parameter and invokes the method with those resolved values. This supports grouped dependency injection when several collaborators belong together in one initialization method. Spring's reference documentation explicitly states that an `@Autowired`-annotated method can have an arbitrary number of arguments.

Also, `@Autowired` is required by default (`required = true`). Therefore, when Spring cannot resolve a required dependency, bean creation fails with an `UnsatisfiedDependencyException`. This exception is part of Spring's bean-creation exception hierarchy and is a runtime exception, so the statement that Spring throws a RuntimeException is accurate. A dependency can be made optional only by explicitly changing the required setting or by using supported optional-dependency patterns, such as `Optional<T>` or `ObjectProvider<T>`. See the Spring Framework reference documentation on Autowired annotation-based injection and the API documentation for UnsatisfiedDependencyException.

QUESTION NO: 3

Which two statements about pointcut expressions are true? (Choose two.)

A. A pointcut expression cannot specify the type of parameters.

B. A pointcut expression will throw an exception if no methods are matched.

C. A pointcut expression cannot have a wildcard for a method name.

D. A pointcut expression can include operators such as the following: && (and), || (or), ! (not).

E. A pointcut expression can be used to select join points which have been annotated with a specific annotation.

ANSWER: D E

Explanation:

Pointcut expressions in Spring AOP use AspectJ-style pointcut designators to define the join points, typically method-execution join points, where advice should apply. They can be combined using logical operators: `&&` requires both constituent expressions to match, `||` matches when either expression matches, and `!` negates an expression. This makes it possible to build precise selections from reusable pointcut conditions, such as restricting an execution pattern to a particular package while excluding a named type or method pattern.

Pointcut expressions can also select join points based on annotations. For example, the `@annotation` designator matches execution of methods carrying a specified annotation, while related designators such as `@within` and `@target` can constrain matching according to annotations on the declaring or runtime target type. Annotation-based pointcuts are commonly used to apply cross-cutting behavior declaratively, such as transaction, auditing, authorization, or monitoring advice, without hard-coding individual method names. Spring documents the supported AspectJ pointcut designators and their composition rules in its AOP reference documentation: Spring Framework: Pointcut Expressions. The AspectJ programming guide also describes the semantics of combining pointcuts and annotation-based matching: AspectJ Language Semantics.

QUESTION NO: 4

Given an ApplicationContext containing three bean definitions of type Foo with bean ids foo1, foo2, and foo3, which three @Autowired scenarios are valid and will allow the ApplicationContext to initialize successfully? (Choose three.)

A. @Autowired public void setFoo (Foo foo) {…}

B. @Autowired @Qualifier("foo3") Foo foo;

C. @Autowired public void setFoo (@Qualifier("foo1") Foo foo) {…}

D. @Autowired private Foo foo;

E. @Autowired private Foo foo2;

F. @Autowired public void setFoo(Foo foo2) {…}

ANSWER: B C E

Explanation:

When several beans share the same type, an `@Autowired` injection point must provide enough information for Spring to select exactly one candidate. `@Autowired @Qualifier("foo3") Foo foo;` is valid because the qualifier narrows the eligible `Foo` beans to the bean whose qualifier or bean name is `foo3`. Likewise, placing `@Qualifier("foo1")` on the method parameter explicitly identifies the `foo1` bean for that setter method.

`@Autowired private Foo foo2;` is also valid because Spring can use the injection-point field name as a fallback candidate name when type-based resolution finds multiple beans and no stronger selection mechanism is present. The field name `foo2` matches the bean id `foo2`, allowing Spring to resolve a single `Foo` dependency and initialize the context. These mechanisms preserve type-driven autowiring while making the required selection unambiguous. Spring documents qualifier-based narrowing and bean-name fallback for autowired dependencies in its Autowired and Qualifier documentation and describes annotation-based dependency injection in the Spring Framework reference.

QUESTION NO: 5

Which statement describes the default rollback behavior for a method annotated with @Transactional? (Choose the best answer.)

A. The transaction rolls back by default for RuntimeException and Error.

B. The transaction rolls back by default for every checked exception.

C. The transaction is committed only when the method returns null.

D. The transaction always rolls back unless @Commit is declared.

ANSWER: A

Explanation:

By default, Spring marks a declarative transaction for rollback when the transactional method throws an unchecked exception, such as a `RuntimeException` or an `Error`. Checked exceptions do not trigger rollback by default. This convention permits application code to distinguish exceptions that normally represent unrecoverable failures from checked exceptions that may be handled as part of normal business processing.

The default can be changed using the `rollbackFor` and `noRollbackFor` attributes of `@Transactional`. For example, `rollbackFor = Exception.class` can request rollback for checked exceptions. See the Spring Framework rollback rules reference and the Transactional Javadoc.

QUESTION NO: 6

Which statement describes the @AfterReturning advice type? (Choose the best answer.)

A. The advice is invoked only if the method returns successfully but not if it throws an exception.

B. The @AfterReturning advice allows behavior to be added after a method returns even if it throws an exception.

C. The advice has complete control over the method invocation; it could even prevent the method from being called at all.

D. Typically used to prevent any exception, thrown by the advised method, from propagating up the call-stack.

ANSWER: A

Explanation:

The advice is invoked only if the method returns successfully but not if it throws an exception. In Spring AOP, `@AfterReturning` is an annotation-based form of after-returning advice: Spring invokes it after a matched method invocation completes normally. “Normally” means that the target method reached its return point and did not exit by throwing an exception. This makes the advice appropriate for work that depends on a successful outcome, such as recording an audit event, updating metrics, logging a successful operation, or inspecting the value returned by the target method.

An `@AfterReturning` advice method can optionally declare a `returning` attribute and a corresponding parameter to receive the returned value. Spring will then run the advice only when the returned value is compatible with that parameter type. The advice executes in the normal-return path and does not serve as exception-handling logic. Spring’s reference documentation describes after-returning advice as advice that runs when a matched method execution returns normally and documents how the `returning` attribute exposes the return value. See Spring Framework reference: @AspectJ advice and Spring API: AfterReturningAdvice.

QUESTION NO: 7

Which three statements are advantages of using Spring’s Dependency Injection? (Choose three.)

A. Dependency injection can make code easier to trace because it couples behavior with construction.

B. Dependency injection reduces the start-up time of an application.

C. Dependencies between application components can be managed external to the components.

D. Configuration can be externalized and centralized in a small set of files.

E. Dependency injection creates tight coupling between components.

F. Dependency injection facilitates loose coupling between components.

ANSWER: C D F

Explanation:

Spring Dependency Injection separates an object's responsibilities from the work of locating, constructing, and wiring the objects it needs. Therefore, dependencies between application components can be managed external to the components: a Spring container creates collaborating objects and supplies their required dependencies through constructors, setters, or fields. This allows application classes to focus on their business behavior rather than on object creation and lookup.

Configuration can also be externalized and centralized in a small set of files. Spring supports Java-based configuration, annotations, and external property sources, enabling deployment-specific values and wiring choices to be maintained outside application logic. This makes changes to environment settings and component composition more manageable.

Dependency injection facilitates loose coupling between components because a component can depend on an interface or abstraction rather than instantiate a concrete collaborator directly. Implementations can consequently be substituted with minimal impact on the consuming component, which improves testability and supports clearer separation of concerns. Spring's reference documentation describes the container as responsible for instantiating, configuring, and assembling application objects through dependency injection. See the Spring Framework IoC Container documentation and the Spring dependency injection documentation.

QUESTION NO: 8

Refer to the exhibit.

```
public interface CustomerRepository extends CrudRepository<Customer, Long>{
}
```

Which statement is true? (Choose the best answer.)

A. CustomerRepository should be a class, not an interface.

B. JPA annotations are required on the Customer class to successfully use Spring Data JDBC.

C. An implementation of this repository can be automatically generated by Spring Data JPA.

D. A class that implements CustomerRepository must be implemented and declared as a Spring Bean.

ANSWER: C

Explanation:

An implementation of this repository can be automatically generated by Spring Data JPA. Spring Data's repository abstraction is specifically designed to create repository implementations at runtime from an interface declaration. When an application uses Spring Data JPA and repository scanning is enabled, an interface extending a supported Spring Data repository interface, such as `CrudRepository`, supplies the domain type and identifier type through its generic parameters. Spring Data JPA then provides the standard persistence operations—including saving entities, looking them up by identifier, retrieving collections, and deleting entities—without requiring application code to write a concrete implementation for those operations.

This convention-based approach is central to Spring Data JPA: the repository interface is registered as a Spring bean and is backed by a proxy implementation created by the framework. Additional query methods can also be derived from method names or defined with query annotations when needed. The domain object must be mapped as a JPA entity for JPA

persistence, while the repository itself remains an interface rather than a manually implemented bean. See the Spring Data JPA reference documentation on core repository concepts and the Spring Framework guide to accessing data with JPA.