MockCert

Study guide | Broadcom | 2V0-72.22

Spring Certified Professional

Notes for every domain of the Spring Certified Professional exam, organized by the official exam guide's domains and weights.

By MockCert about 70 min read Updated November 13, 2026

CONTENTS

DOMAIN 1 · 32%
Spring Core 2

DOMAIN 2 · 16%
Data Management 19

DOMAIN 3 · 10%
Spring MVC 28

DOMAIN 4 · 10%
Testing 34

DOMAIN 5 · 16%
Security 40

DOMAIN 6 · 16%
Spring Boot 49

MockCert is an independent certification preparation platform. It is not affiliated with, endorsed by or operated by Broadcom. Exam domains and weights follow the official exam guide. These notes were last updated on November 13, 2026; the notes at mockcert.com are kept up to date as the exam changes.

Domain 1 | 32% of topics | 23 min read

Spring Core

Overview

Every point below comes from Broadcom's official documentation; each section lists its sources.

Introduction to Spring Framework

Start here

THINK FIRST

What does the Spring IoC container do for your application, and what do AOP and data access support add to it?

Why it matters: Questions ask what the container, beans, ApplicationContext and Spring's core services each do in an application.

Spring's IoC container creates, configures and assembles your objects, called beans. With dependency injection, objects declare what they need and the container supplies it. Optional AOP adds crosscutting services, and data access support translates exceptions.

Key terms

Term	Meaning
Bean	An object that a Spring IoC container instantiates, assembles and manages.
Dependency injection	Objects declare dependencies as constructor, factory method or property arguments, and the container injects them.
ApplicationContext	The interface that represents the Spring IoC container and builds the beans.
Aspect	The AOP unit of modularity; it packages a concern that cuts across many types.
DataAccessException	The root of Spring's exception hierarchy for translated data access errors.

The IoC container

ApplicationContext is the container that builds your beans. It instantiates, configures and assembles beans. Its foundation is two packages: `org.springframework.beans` and `org.springframework.context`. ApplicationContext is a full superset of BeanFactory, adding enterprise-specific features to its basic configuration framework.

Dependency injection

Objects declare dependencies; the container injects them. A dependency arrives through constructor arguments, factory method arguments or properties set after construction. The container injects them when it creates the bean. Your code should ideally never call `getBean()`, so it stays free of Spring APIs.

EXAM TRAP

Looking beans up with `getBean()` throughout your code is not the goal. Ideal application code has no `getBean()` calls and no dependency on Spring APIs.

Configuration metadata

You describe beans with annotations, Java classes or XML. Metadata can take three forms: annotated component classes, configuration classes whose factory methods create beans, or external XML files and Groovy scripts. Java configuration uses @Bean methods inside a @Configuration class, one per bean definition. Fine-grained domain objects are usually left out, because repositories and business logic create them.

Creating a context

Stand-alone apps create a context; Spring Boot does it for you. Stand-alone applications typically create an AnnotationConfigApplicationContext or a ClassPathXmlApplicationContext. In Spring Boot, the context is bootstrapped implicitly from common setup conventions. When you must, retrieve a bean by calling `getBean()` with its name and required type. The context also provides internationalization (i18n) support.

Events are synchronous

By default, listeners receive events synchronously. `publishEvent()` blocks until every listener has finished with the event. You can still register as many listeners as you like.

EXAM TRAP

Publishing an event does not return at once by default. `publishEvent()` waits until all listeners have processed it.

AOP in brief

In AOP, the aspect is the unit of modularity, not the class. Aspects modularize concerns such as transaction management that cut across many types and objects. Declarative transaction management is the most important enterprise service built this way. You write aspects in a schema-based style or the @AspectJ style. The IoC container does not depend on AOP, so it is optional.

Data access exceptions

Spring translates technology-specific exceptions for you. An SQLException becomes part of a hierarchy rooted at DataAccessException. JPA- and Hibernate-specific exceptions turn into focused runtime exceptions too. The translated exceptions wrap the original, so no error detail is lost.

The @Repository annotation

@Repository guarantees exception translation for DAOs. It is the best way to make sure DAOs and repositories translate exceptions. It also lets component scanning find and configure them without XML entries. Spring's DAO support lets you switch between JDBC, Hibernate and JPA fairly easily.

EXAM TRAP

@Repository is more than a marker for component scanning. It is also the best way to guarantee that your DAOs and repositories translate exceptions.

Key takeaways

- A bean is any object that the Spring IoC container instantiates, assembles and manages.
- ApplicationContext is the IoC container and a full superset of BeanFactory.
- By default, event listeners run synchronously and `publishEvent()` blocks until they finish.
- Use @Repository so DAOs get exception translation into DataAccessException.

Sources: Container Overview :: Spring Framework (opens in a new tab) · Introduction to the Spring IoC Container and Beans :: Spring Framework (opens in a new tab) · Aspect Oriented Programming with Spring :: Spring Framework (opens in a new tab) · DAO Support :: Spring Framework (opens in a new tab) · Additional Capabilities of the ApplicationContext :: Spring Framework (opens in a new tab)

Java Configuration

Start here

THINK FIRST

How do you define beans in Java code, wire them together and control how many instances Spring creates?

Why it matters: Questions show configuration classes and ask which beans exist, what they are named, how they connect and how long they live.

A `@Configuration` class declares beans with `@Bean` methods, and Spring registers each one in the application context. A CGLIB subclass makes calls between those methods return the container's beans. Beans are singletons by default, and `@Scope` changes that.

Key terms

Term	Meaning
@Bean	A method-level annotation that declares a bean; the direct Java analog of the XML bean element.
Lite mode	`@Bean` methods outside a full `@Configuration` class; direct calls to them are not intercepted.

Term	Meaning
Bean scope	Set on a `@Bean` method with `@Scope`; singleton is the default, and a singleton is per container and per bean.
CGLIB subclass	The startup-time subclass Spring creates for each `@Configuration` class to intercept `@Bean` calls.
Prototype scope	A non-singleton scope chosen with `@Scope`; Spring builds such a bean on request, not when the container starts.

Declaring beans with @Bean

Each `@Bean` method declares one bean. By default the bean takes the method's name. You can put `@Bean` methods in a `@Configuration` class or a `@Component` class. Declare the most specific return type when the bean implements several interfaces.

EXAM TRAP

The bean is not named after its return type. By default it gets the name of the `@Bean` method.

Bootstrapping the context

Pass configuration classes to `AnnotationConfigApplicationContext`. Spring registers the class itself and each of its `@Bean` methods as bean definitions. You can also use the no-arg constructor and then call `register()`. Stand-alone apps commonly use this context or `ClassPathXmlApplicationContext`.

Getting beans out of the context

`getBean()` **returns a bean by its name and required type.** The `ApplicationContext` offers this method for retrieving bean instances. Ideally, your application code never calls `getBean()` at all, so it has no dependency on Spring APIs.

Splitting configuration

`@Import` **loads `@Bean` definitions from another configuration class.** It plays the same role as the import element in Spring XML. Configuration classes are meta-annotated with `@Component`, so component scanning finds them too. Reader delegates let one context mix several configuration formats.

Wiring dependencies

Declare a dependency as a `@Bean` method parameter. A `@Bean` method can take any number of parameters. Prefer them over `@Autowired` fields in a configuration class, which is processed early and may force early initialization. Use constructors for mandatory dependencies and setters for optional ones.

EXAM TRAP

Two beans that need each other through constructor injection do not wire up. Spring detects the cycle at runtime and throws `BeanCurrentlyInCreationException`.

Full versus lite mode

Only full `@Configuration` classes intercept `@Bean` calls. Spring subclasses every `@Configuration` class with CGLIB, so calling another `@Bean` method returns the cached bean. For the same reason, a configuration class must not be final.

Full configuration mode compared with lite mode

Feature	Full `@Configuration`	Lite mode
Where it applies	Classes annotated with `@Configuration`	Other classes, or `@Configuration(proxyBeanMethods=false)`
Calling another `@Bean` method	The container returns the cached bean	A plain Java call that creates a new instance
Inter-bean dependencies by method call	Supported	Not supported in plain `@Component` classes

EXAM TRAP

In a `@Component` class, calling one `@Bean` method from another does not reuse the singleton. Each call creates a new instance.

Bean scopes

Singleton is the default scope. A singleton is per container and per bean. Change the scope on a `@Bean` method with `@Scope`. Spring supports **6** scopes, and **4** of them need a web-aware `ApplicationContext`.

Singleton and prototype beans compared

Feature	Singleton	Prototype
Default scope	Yes	No; set it with `@Scope`
When created	With the container, when pre-instantiated (the default)	Only when requested
Initialization callbacks	Called	Called

EXAM TRAP

Prototype beans do not get their configured destruction callbacks, even though their initialization callbacks run.

Key takeaways

- By default, a bean declared with `@Bean` is named after its method.
- Inter-bean `@Bean` calls reuse the container's bean only inside a full `@Configuration` class.
- Prefer `@Bean` method parameters over `@Autowired` fields in configuration classes.

- Prototype beans get their initialization callbacks but not their configured destruction callbacks.

Sources: Instantiating the Spring Container by Using AnnotationConfigApplicationContext :: Spring Framework (opens in a new tab) · Using the @Configuration annotation :: Spring Framework (opens in a new tab) · Using the @Bean Annotation :: Spring Framework (opens in a new tab) · Basic Concepts: @Bean and @Configuration :: Spring Framework (opens in a new tab) · Bean Scopes :: Spring Framework (opens in a new tab) · Dependency Injection :: Spring Framework (opens in a new tab) · Container Overview :: Spring Framework (opens in a new tab) · Composing Java-based Configurations :: Spring Framework (opens in a new tab)

Properties and Profiles

Start here

THINK FIRST

How can one Spring application run with different settings in development and production without changing its code?

Why it matters: Questions ask which property value wins, which beans load under a given profile, or what a SpEL expression does.

Spring Boot reads properties from several places in a fixed order, and later ones override earlier ones. Profiles switch whole parts of the configuration on or off per environment. SpEL works on an object graph at runtime, and bean definitions can use it through `#{ }` expressions.

Key terms

Term	Meaning
Profile	A named part of the configuration that loads only when that profile is active.
spring.profiles.active	The Environment property that lists which profiles are active.
SpEL	Spring Expression Language: queries and changes an object graph at runtime, with method calls and string templating.
Evaluation context	The setting a SpEL expression runs in; it controls which parts of the language are allowed.

Getting property values into beans

Inject, read from the Environment, or bind to an object. `@Value` puts one value straight into a bean. The Environment abstraction reads values in code. `@ConfigurationProperties` binds a group of values to a structured object.

Which value wins

Later entries in Boot's fixed order override earlier ones. Command-line options such as `--server.port=9000` become properties and beat file-based settings. If `.properties` and YAML files sit

in the same location, `.properties` wins. Environment variables may use underscores, as in `SPRING_CONFIG_NAME`.

EXAM TRAP

Property files added by annotation join the Environment only when the context refreshes. That is too late for `logging.*` and `spring.main.*` settings.

Limiting beans to an environment

Mark a component or configuration class with `@Profile`. A profile limits part of the configuration to the environments that need it. Any `@Component`, `@Configuration` or `@ConfigurationProperties` class can carry `@Profile`. A profile group gives one logical name to several related profiles.

Activating profiles

Set `spring.profiles.active`, or fall back to `default`. With no active profile, a profile named `default` is enabled. Its name can be changed with `spring.profiles.default`.

Profile properties compared

Property	What it does	Worth knowing
`spring.profiles.active`	Names the active profiles	A command-line switch can replace the file value
`spring.profiles.include`	Adds profiles	Adds to the active ones instead of replacing them
`spring.profiles.default`	Renames the fallback profile	Used only when no profile is active

Conditions on properties

`@ConditionalOnProperty` **includes configuration based on a property.** By default, it matches any property that exists and is not `false`. When the `name` attribute lists several properties, every one must pass. `@ConditionalOnExpression` uses a SpEL result instead.

EXAM TRAP

Referencing a bean inside a `@ConditionalOnExpression` expression initializes it very early. That bean then misses post-processing such as configuration properties binding.

SpEL in bean definitions

Wrap each expression in `#{ }` delimiters. Every bean in the context is available by its name, including `environment`, `systemProperties` and `systemEnvironment`. Whitespace between tokens is ignored unless it is inside a string literal.

Evaluation contexts and limits

Choose the evaluation context by how far you trust the expression. Parsing can throw `ParseException`; `getValue()` can throw `EvaluationException`. By default an expression is capped at **10,000** characters and **10,000** operations.

SpEL evaluation contexts compared

Feature	SimpleEvaluationContext	StandardEvaluationContext
Language coverage	A subset of SpEL syntax	The complete language
Type references, constructors, bean references	Excluded	Available
Untrusted expressions	Built for limited syntax	Must never evaluate them

The SpEL compiler

The compiler is off by default, so expressions are interpreted. `spring.expression.compiler.mode` can be `off`, `immediate` or `mixed`. In immediate mode, a failing compiled expression throws to the caller. Expressions with assignment or using the conversion service cannot be compiled.

Key takeaways

- Command-line properties override values from application files.
- `spring.profiles.include` adds profiles; `spring.profiles.active` sets them.
- With no active profile, Spring enables the profile named `default`.
- Never evaluate untrusted expressions with `StandardEvaluationContext`.

Sources: Externalized Configuration :: Spring Boot (opens in a new tab) · Profiles :: Spring Boot (opens in a new tab) · Spring Expression Language (SpEL) :: Spring Framework (opens in a new tab) · Expressions in Bean Definitions :: Spring Framework (opens in a new tab) · Evaluation :: Spring Framework (opens in a new tab) · Creating Your Own Auto-configuration :: Spring Boot (opens in a new tab) · Language Reference :: Spring Framework (opens in a new tab)

Annotation-Based Configuration and Component Scanning

Start here

THINK FIRST

How does Spring decide which annotated classes become beans, and when their lifecycle methods run?

Why it matters: Questions show annotated classes and ask which ones scanning detects, what each bean is called and when callbacks run.

Component scanning finds classes marked with a stereotype annotation and registers each one as a bean.

With `CommonAnnotationBeanPostProcessor` registered, `@PostConstruct` and `@PreDestroy` methods then run at the matching lifecycle points.

Key terms

Term	Meaning
Component scanning	Searching a base package for stereotype-annotated classes and registering them as beans.
Stereotype annotation	`@Component` or one of its role-specific forms: `@Repository`, `@Service` or `@Controller`.
CommonAnnotationBeanPostProcessor	The processor that recognizes the JSR-250 lifecycle annotations `@PostConstruct` and `@PreDestroy`.
JSR-330	The standard Dependency Injection for Java annotations, such as `@Inject` and `@Named`.

Turning on component scanning

Add `@ComponentScan` to a `@Configuration` class. Set its `basePackages` attribute to a common parent package of your components. In XML, the component-scan element also switches on annotation config, so you rarely need both. An `AnnotationConfigApplicationContext` already registers the annotation post-processors for you.

What scanning detects

Only stereotype-annotated classes are detected by default. Default candidates carry `@Component`, `@Repository`, `@Service`, `@Controller` or `@Configuration`. A custom annotation also counts when it is itself annotated with `@Component`. Without an explicit name, the bean gets the uncapitalized simple class name.

> **EXAM TRAP**
> A plain class in the scanned package is not registered just because of where it sits. By default it needs a stereotype annotation.

Choosing a stereotype

Pick the stereotype that matches the class's layer. `@Repository`, `@Service` and `@Controller` specialize `@Component` for the persistence, service and presentation layers. For a service class, `@Service` is the better choice than `@Component`.

Stereotype annotations compared by layer and extra role

Annotation	Layer	Extra role
`@Component`	Any, generic	Base stereotype; can mark custom annotations
`@Repository`	Persistence (DAO)	Enables automatic exception translation

Annotation	Layer	Extra role
`@Service`	Service	Preferred over `@Component` for service classes
`@Controller`	Presentation	Combined with `@ResponseBody` in `@RestController`

Lifecycle annotations

`@PostConstruct` and `@PreDestroy` are JSR-250 callbacks. With `CommonAnnotationBeanPostProcessor` registered, they run at the same point as Spring's lifecycle interfaces or declared callback methods. Classes defined with `@Bean` can use them too. Since **Jakarta EE 9**, they live in `jakarta.annotation`.

EXAM TRAP

Since Jakarta EE **9**, these annotations live in `jakarta.annotation`, not `javax.annotation`. The old package left the core JDK in **JDK 9** and was removed in **JDK 11**.

Inferred destroy methods

A public `close` or `shutdown` method runs at shutdown. Beans defined with Java configuration get this destruction callback by default when they have such a method. Add `@Bean(destroyMethod = "")` if the container should not call it.

Mixing annotations and XML

XML property values override annotation injection. Spring applies annotation injection before external property injection. So when both wire the same property, the XML value is the one that stays. Spring Boot favors Java configuration, ideally one main `@Configuration` class.

EXAM TRAP

Annotations do not beat XML. They are applied first, so an XML-specified property value replaces them.

JSR-330 alternatives

Spring supports `@Inject` and `@Named` from `jakarta.inject`. These are standard JSR-330 annotations. The table compares each one with the Spring annotation for the same job.

Spring annotations and their JSR-330 counterparts

Feature	Spring	JSR-330
Inject a dependency	`@Autowired`	`@Inject`, with no `required` attribute
Declare a component	`@Component`, composable	`@Named`, not composable
Inject a property value	`@Value`	No equivalent

Feature	Spring	JSR-330
Default scope in Spring	Singleton	Singleton, though JSR-330's own default is like prototype

Key takeaways

- Scanning detects `@Component`, its stereotypes and custom annotations meta-annotated with `@Component`.
- Use `@Service` for service classes and `@Repository` for DAOs, which adds exception translation.
- XML property values override annotations, because annotation injection happens first.
- `@PostConstruct` and `@PreDestroy` now come from the `jakarta.annotation` package.

Sources: Classpath Scanning and Managed Components :: Spring Framework (opens in a new tab) · Using @PostConstruct and @PreDestroy :: Spring Framework (opens in a new tab) · Annotation-based Container Configuration :: Spring Framework (opens in a new tab) · Using the @Bean Annotation :: Spring Framework (opens in a new tab) · Configuration Classes :: Spring Boot (opens in a new tab) · Using JSR-330 Standard Annotations :: Spring Framework (opens in a new tab)

Spring Bean Lifecycle

Start here

THINK FIRST

How does Spring turn bean definitions into ready, possibly proxied objects, and where can your code hook into that process?

Why it matters: Questions describe a bean setup and ask which callback, post-processor or proxy acts, and in what order.

Spring first lets a `BeanFactoryPostProcessor` edit bean definitions, then creates the beans. Each bean gets its dependencies and then its init callbacks, with every `BeanPostProcessor` called before and after them. A post-processor may hand back a proxy that adds behavior.

Key terms

Term	Meaning
BeanFactoryPostProcessor	Reads and may change bean configuration metadata before ordinary beans are created.
BeanPostProcessor	Gets a callback for each new bean, once before and once after its init callbacks.
Initialization callback	A method Spring calls once a bean has all its dependencies, such as a `@PostConstruct` method.
CGLIB proxy	The kind of proxy Spring AOP needs to proxy a class rather than an interface, used by default when there is no interface.

Term	Meaning
@Primary	Marks the preferred bean when several beans match a single-valued injection point.

Bean definitions come first

A `BeanFactoryPostProcessor` edits metadata before beans exist. It reads bean definitions and can change them before the container creates any other bean. Spring ships ready-made ones, such as `PropertySourcesPlaceholderConfigurer` and `PropertyOverrideConfigurer`. Register your own from a `static @Bean` method. That avoids clashes with `@Autowired`, `@Value` and `@PostConstruct` processing in the configuration class.

EXAM TRAP

A `BeanFactoryPostProcessor` works on bean definitions, not on created bean objects. Code that inspects or wraps instances belongs in a `BeanPostProcessor`.

Hooks around each bean

A `BeanPostProcessor` runs before and after init callbacks. Spring calls it for every bean it creates. The first call comes before `afterPropertiesSet()` or a declared init method. The second comes after all init callbacks. It may check for callback interfaces or wrap the bean in a proxy.

The two post-processor types compared

Feature	BeanFactoryPostProcessor	BeanPostProcessor
Works on	Bean configuration metadata	Each bean instance the container creates
When it runs	Before any other beans are instantiated	Before and after each bean's init callbacks
Typical use	Placeholder and property override configurers	Checking callback interfaces or wrapping a bean in a proxy

Init callback order

`@PostConstruct` runs first, then `afterPropertiesSet()`, then `init()`. This order applies when one bean has several mechanisms with different method names. Each callback runs right after the bean receives all its dependencies. `afterPropertiesSet()` comes from `InitializingBean`. In Java configuration, name a plain init method with the `initMethod` attribute of `@Bean`.

EXAM TRAP

Init callbacks run on the raw bean, not on its proxy. AOP interceptors are not yet applied when the init callback runs.

Annotations and destroy methods

Prefer `@PostConstruct` and `@PreDestroy` over Spring interfaces. They keep beans free of Spring-specific interfaces. They work when `CommonAnnotationBeanPostProcessor` is registered. `DisposableBean` instead gives a `destroy()` method. For `@Bean` methods, Spring infers a public `close` or `shutdown` method as the destroy method by default.

EXAM TRAP

The old `javax.annotation` package was split out in JDK **9** and removed in JDK **11**. As of Jakarta EE **9**, these annotations live in `jakarta.annotation`.

How proxies add behavior

Auto-proxying is itself a `BeanPostProcessor`. It wraps a bean in a proxy that applies the aspects. For a class with no interface, Spring AOP uses a CGLIB proxy by default. CGLIB is repackaged inside `spring-core`, so you add nothing to the classpath.

EXAM TRAP

A `BeanPostProcessor`, and any bean it references directly, cannot be auto-proxied. No aspects are woven into those beans.

Proxied configuration classes

`@Configuration` **classes are CGLIB subclasses at startup.** The subclass checks the container for a cached bean before calling your `@Bean` method. So one `@Bean` method can call another and still receive the container's bean. That works only in `@Configuration` classes, not plain `@Component` classes. Such classes must not be `final`. With `proxyBeanMethods = false`, those calls are not intercepted, so use injection.

Bean creation order

`@DependsOn` **makes named beans initialize first.** The `depends-on` attribute has the same effect. Separate several bean names with commas, whitespace or semicolons. The dependent bean starts after its dependency and stops before it.

EXAM TRAP

The destruction ordering from `depends-on` covers singleton beans only. The initialization ordering is not limited that way.

Injecting by type safely

Use `@Primary` or `@Qualifier` when several beans share a type. `@Primary` marks the preferred candidate when a single bean must be injected. When exactly one candidate carries it, that bean wins. `@Qualifier` values narrow the type matches to one bean. Since Spring **6.2**, `@Fallback` marks backup beans, so one remaining regular bean acts as primary.

EXAM TRAP

`@Qualifier` does not replace matching by type. It narrows the set of beans that already match the type down to one.

Key takeaways

- A `BeanFactoryPostProcessor` changes bean definitions; a `BeanPostProcessor` works on each created bean.
- With different method names, init order is `@PostConstruct`, then `afterPropertiesSet()`, then the custom init method.
- For a class without an interface, Spring AOP uses a CGLIB proxy by default.
- Declare a `@Bean` method that returns a `BeanFactoryPostProcessor` as `static`.
- Use `@Primary` or `@Qualifier` when several beans match one type.

Sources: Container Extension Points :: Spring Framework (opens in a new tab) · Customizing the Nature of a Bean :: Spring Framework (opens in a new tab) · Using @PostConstruct and @PreDestroy :: Spring Framework (opens in a new tab) · AOP Proxies :: Spring Framework (opens in a new tab) · Fine-tuning Annotation-based Autowiring with @Primary or @Fallback :: Spring Framework (opens in a new tab) · Using the @Configuration annotation :: Spring Framework (opens in a new tab) · Using depends-on :: Spring Framework (opens in a new tab) · Fine-tuning Annotation-based Autowiring with Qualifiers :: Spring Framework (opens in a new tab)

Aspect Oriented Programming

Start here

THINK FIRST

How do you choose the right advice type and pointcut when you add a cross-cutting behavior with Spring AOP?

Why it matters: Questions describe a cross-cutting need and ask which advice type, pointcut or proxy behavior applies to it.

An aspect packages behavior that cuts across many services, such as retrying failed operations. Spring wraps advised beans in proxies and runs advice at method execution join points. Pick the least powerful advice that does the job.

Key terms

Term	Meaning
Aspect	A bean whose class carries `@Aspect`, which Spring uses to configure AOP; suited to a requirement that cuts across many services.
Advice	The code an aspect runs at a matched join point: before it, after it or around it.

Term	Meaning
Join point	A point where advice can run; in Spring AOP it is always a method execution on a Spring bean.
Pointcut	An expression that selects join points, declared with @Pointcut on a method that returns void.
Auto-proxying	Spring generating a proxy for an advised bean so it can intercept calls and run the advice.

Why aspects exist

Aspects hold behavior that cuts across many services. Retrying idempotent operations that fail from concurrency problems is a classic example. Without an aspect, the retry logic would repeat in every service of the service layer. An aspect applies it transparently, so the client never sees the locking failure.

Proxies and join points

Spring AOP is proxy-based and advises only method executions. When a bean is advised by an aspect, Spring generates a proxy that intercepts its method calls. Only method execution join points on Spring beans are supported. The proxy sits outside the object, so a call from inside the same target is not intercepted.

EXAM TRAP

A method that calls another method on the same object does not trigger advice on the inner call. That call never passes through the proxy.

Enabling and detecting aspects

@Aspect marks the class, but scanning also needs @Component. With @AspectJ support on, Spring detects any bean whose class has @Aspect. The `org.aspectj:aspectjweaver` library, version **1.9** or later, must be on the classpath. AspectJ only parses and matches pointcuts; the runtime stays pure Spring AOP, with no AspectJ compiler or weaver.

EXAM TRAP

Putting @Aspect on a class is not enough for component scanning to find it. Add @Component or a qualifying stereotype annotation as well.

Five advice types

Each advice type runs at a different moment of the call. The table compares them. @After is after finally advice and fires for every outcome, while @AfterReturning fires only on a successful return.

The five advice types compared

Advice	Runs when	Key point
@Before	Before the join point runs	Cannot change the return value; an exception stops the chain
@AfterReturning	After a normal, successful return only	Sees the return value but cannot modify it
@AfterThrowing	When the method exits by throwing an exception	Does not run on a normal return
@After	After any outcome, like a finally block	Runs for a normal return and for an exception
@Around	Around the call, given a ProceedingJoinPoint	Returns Object and can call proceed more than once

EXAM TRAP

Choosing @After when you want only successful returns is a mistake. It also runs when an exception is thrown, so use @AfterReturning instead.

Least powerful advice

Pick the weakest advice that meets the need. Do not use around advice when before advice is enough. Before advice never calls `proceed()`, so it cannot forget to continue down the interceptor chain. Around advice fits when you must call proceed several times, as retry logic does.

Pointcut expressions

Parameter patterns set how many arguments a method may take. In an execution expression, `()` matches no parameters, `(..)` matches zero or more, and `(*)` matches exactly one of any type. Declare a reusable pointcut with @Pointcut on a method returning void. The extra `bean` designator limits matching to named Spring beans, with wildcards for a set.

EXAM TRAP

AspectJ designators such as `call`, `get`, `set` or `cflow` are not supported. Using them in a Spring AOP pointcut throws an IllegalArgumentException.

Advice ordering

Within one aspect, @Around has the highest precedence. For advice in the same @Aspect at the same join point, the order is @Around, @Before, @After, @AfterReturning, then @AfterThrowing. An aspect can also implement the Ordered interface. A retry aspect does this to rank above transaction advice, so each retry gets a fresh transaction.

The lower-level advice API

Spring's advice interfaces mirror the same ideas. Around advice classes implement the AOP Alliance MethodInterceptor interface. ThrowsAdvice is a marker interface with no methods of its own. If a

throws-advice method itself throws, its exception replaces the original one the user sees.

Key takeaways

- Spring AOP advises only method executions on Spring beans, through proxies.
- Calls from inside the target object bypass the proxy and its advice.
- Use @After for every outcome and @AfterReturning for successful returns only.
- Choose around advice only when you need control over proceed, such as for retries.

Sources: An AOP Example :: Spring Framework (opens in a new tab) · Enabling @AspectJ Support :: Spring Framework (opens in a new tab) · Declaring a Pointcut :: Spring Framework (opens in a new tab) · Declaring Advice :: Spring Framework (opens in a new tab) · Declaring an Aspect :: Spring Framework (opens in a new tab) · Advice API in Spring :: Spring Framework (opens in a new tab) · @AspectJ support :: Spring Framework (opens in a new tab)

Domain 2 | 16% of topics | 11 min read

Data Management

Overview

Every point below comes from Broadcom's official documentation; each section lists its sources.

Introduction to Spring JDBC

Start here

THINK FIRST

What does JdbcTemplate take off your hands, and what is still your job when you query a database with it?

Why it matters: Questions ask which JdbcTemplate method or callback fits a task, and which exception a failed data access call throws.

`JdbcTemplate` opens and closes JDBC objects such as connections for you, which helps you avoid mistakes such as leaving a connection unclosed. It also turns every `SQLException` into an unchecked `DataAccessException`.

Key terms

Term	Meaning
JdbcTemplate	The central class of the JDBC core package; it creates and releases JDBC connections and related objects.
RowCallbackHandler	A callback interface that pulls values out of each row of a `ResultSet`.
DataAccessException	The root of Spring's data access exception hierarchy; it wraps the original exception.
Exception translator	The component that converts a JDBC `SQLException` into a Spring data access exception.

Creating a JdbcTemplate

Create it inside the DAO, or inject it as a bean. A DAO can instantiate the template directly. It can also receive one configured in the Spring container as a bean reference. Once configured, the template is thread-safe, so one shared instance can serve many DAOs.

EXAM TRAP

You do not need one `JdbcTemplate` per DAO or per thread. A configured instance is thread-safe and can be shared.

What the template handles

Connection handling, transactions and SQL logging come built in. The template creates and releases what it uses, which helps you avoid errors such as a forgotten connection close. It joins Spring transaction support for both JTA and JDBC transactions. It logs every statement at DEBUG level under its class name.

Picking the right method

Each kind of SQL work has its own template method. Match the method to what the statement does and what you need back.

Common JdbcTemplate methods and their use

Method	Use it for	Worth knowing
`update(..)`	Insert, update and delete statements	One method covers all three changes
`execute(..)`	Any SQL, often DDL statements	Runs arbitrary SQL
`queryForObject(..)`	A single value of a given Java class	A bad type conversion throws `InvalidDataAccessApiUsageException`
`queryForList(..)`	Rows as a list of maps	Each map is keyed by column name

Reading results with callbacks

Callbacks decide how each row of a result set is read. A `RowCallbackHandler` extracts values from every row of a `ResultSet`. For a JavaBean class, `BeanPropertyRowMapper.newInstance` builds a row mapper from the class you pass in. These callback interfaces live in `org.springframework.jdbc.core` with the template.

From SQLException to DataAccessException

`JdbcTemplate` **never lets a raw** `SQLException` **escape.** It translates JDBC exceptions into the hierarchy in `org.springframework.dao`, rooted at `DataAccessException`. The original exception stays wrapped inside, so no detail is lost. All translated exceptions are unchecked.

EXAM TRAP

Translated exceptions are unchecked. You are not forced to catch them; catch only the ones you can recover from and let the rest propagate.

Default and custom translators

Since 6.0, the default is `SQLExceptionSubclassTranslator`**.** It checks `SQLException` subclasses from JDBC **4** and falls back to SQLState checks through `SQLStateSQLExceptionTranslator`. For your own translator, call `setExceptionTranslator` on the template. Use that template for all the affected data access.

Translation for repositories

Mark DAOs and repositories with `@Repository`**.** It is the best way to guarantee exception translation. `PersistenceExceptionTranslationPostProcessor` finds every `PersistenceExceptionTranslator` and advises each `@Repository` bean, so thrown exceptions get translated.

Key takeaways

- `JdbcTemplate` creates and releases JDBC connections, which helps you avoid forgetting to close a connection.
- A configured `JdbcTemplate` is thread-safe and can be shared by many DAOs.
- Use `update(..)` for insert, update and delete, and `execute(..)` for DDL.
- `JdbcTemplate` and `JdbcTransactionManager` do not propagate `SQLException`; translated exceptions are unchecked.

Sources: Using the JDBC Core Classes to Control Basic JDBC Processing and Error Handling :: Spring Framework (opens in a new tab) · Package Hierarchy :: Spring Framework (opens in a new tab) · DAO Support :: Spring Framework (opens in a new tab) · General ORM Integration Considerations :: Spring Framework (opens in a new tab) · Simplifying JDBC Operations with the SimpleJdbc Classes :: Spring Framework (opens in a new tab)

Transaction Management with Spring

Start here

THINK FIRST

When does a `@Transactional` method start, join or roll back a transaction, and how do tests change that?

Why it matters: Questions show a transactional method or test and ask which propagation, rollback rule or default applies.

Spring wraps a transactional bean in an AOP proxy that runs transactions around method calls. Propagation decides whether a call joins a running transaction or starts its own. Rollback rules decide which exceptions undo the work.

Key terms

Term	Meaning
Transactional proxy	An AOP proxy that uses a `TransactionInterceptor` and a transaction manager around method calls.
PlatformTransactionManager	The manager imperative transactions need; reactive transactions use `ReactiveTransactionManager` instead.
Propagation	How a method's transaction relates to one already running; the default is `REQUIRED`.
Rollback rule	A setting that marks the transaction for rollback when a matching exception type is thrown.
Rollback-only	A mark that forces rollback; an outer caller that still commits gets `UnexpectedRollbackException`.

How declarative transactions work

AOP proxies apply transactions from XML or annotation metadata. The proxy pairs a `TransactionInterceptor` with a transaction manager around each call. In Java configuration, `@EnableTransactionManagement` switches this on. Methods returning `Publisher` or Kotlin `Flow` get reactive management; all other return types, `void` included, get imperative management.

> **EXAM TRAP**
>
> In proxy mode, the default, a call from one method to another in the same object skips the proxy. That self-invocation runs with no transaction, even if the called method has `@Transactional`.

Defaults and overrides

A method-level `@Transactional` beats the class-level settings. The most derived location wins. The XML `tx:advice` defaults are isolation `DEFAULT` and a read-write transaction. Its timeout is the one the underlying system uses, or no timeout where that system lacks them. Read-only transactions suit code that only reads data and can help performance, for example with Hibernate.

Propagation choices

`REQUIRES_NEW` **always starts its own physical transaction.** `REQUIRED`, the default, joins an outer transaction. A joining scope takes the outer scope's settings and silently ignores its own isolation, timeout or read-only flag. Spring offers every propagation option known from EJB CMT.

Three propagation settings compared

Feature	REQUIRED	REQUIRES_NEW	NESTED
Physical transaction	Joins the outer one and takes on its settings	An independent one for each affected scope	A single one, with savepoints to roll back to
Default propagation	Yes	No	No

Feature	REQUIRED	REQUIRES_NEW	NESTED
Uses the isolation attribute	Yes	Yes	No

EXAM TRAP

When an inner scope silently marks the shared transaction rollback-only, the outer commit does not go through. The outer caller gets an `UnexpectedRollbackException`.

Default rollback behavior

By default, unchecked exceptions roll back and checked ones do not. Any `RuntimeException` or `Error` triggers rollback. Since Spring **6.2**, `rollbackOn=ALL_EXCEPTIONS` on `@EnableTransactionManagement` makes checked exceptions roll back too. Spring's own `TransactionException` is unchecked, because it extends `RuntimeException`.

EXAM TRAP

A checked exception does not roll back by default. Since Spring **6.2**, `rollbackOn=ALL_EXCEPTIONS` can change that default for all exceptions.

Custom rollback rules

A type-based rollback rule also matches subclasses. A thrown exception matches when it is the configured type or a subclass of it. If several rules match, Spring applies the strongest one. In XML, `rollback-for` and `no-rollback-for` each take a comma-delimited list of exceptions.

Threads and transaction managers

A thread-bound transaction does not reach new threads. `@Transactional` usually relies on a `PlatformTransactionManager` binding the transaction to the current thread. Threads started inside the method do not share it. `getTransaction()` returns a `TransactionStatus` for a new or existing transaction. `JtaTransactionManager` uses the container's global infrastructure, so it needs no `DataSource`.

Transactions in tests

Transactional test methods roll back by default. The TestContext Framework rolls back even when no `@Rollback` is declared. A method-level `@Rollback` can override a class-level `@Rollback` or `@Commit`. An `@AfterTransaction` void method runs once the test's transaction ends, and it need not be public.

EXAM TRAP

Leaving out `@Rollback` does not make a test commit. Rollback is already the default for transactional integration tests.

Key takeaways

- `@Transactional` works through an AOP proxy, so self-invocation gets no transaction in proxy mode.
- The default propagation is `REQUIRED`; `REQUIRES_NEW` always uses an independent physical transaction.
- By default, `RuntimeException` and `Error` cause rollback, while checked exceptions do not.
- Rollback rules match the configured exception type and its subclasses; the strongest match wins.
- Transactional test methods roll back by default, even without `@Rollback`.

Sources: Understanding the Spring Framework's Declarative Transaction Implementation :: Spring Framework (opens in a new tab) · Using @Transactional :: Spring Framework (opens in a new tab) · Transaction Propagation :: Spring Framework (opens in a new tab) · Rolling Back a Declarative Transaction :: Spring Framework (opens in a new tab) · <tx:advice/> Settings :: Spring Framework (opens in a new tab) · Understanding the Spring Framework Transaction Abstraction :: Spring Framework (opens in a new tab) · @Rollback :: Spring Framework (opens in a new tab) · @AfterTransaction :: Spring Framework (opens in a new tab)

Spring Boot and Spring Data for Backing Stores

Start here

THINK FIRST

In a Spring Boot JPA application, what does Boot set up for you, and what do you still declare yourself?

Why it matters: Questions show a JPA setup, repository or data test and ask what Spring Boot configures or how a query is built.

Add the Data JPA starter and Spring Boot brings Hibernate, a connection pool and entity scanning. You declare repository interfaces, and Spring Data builds queries from their method names. In tests, `@DataJpaTest` sets up entities and repositories for you.

Key terms

Term	Meaning
Data JPA starter	`spring-boot-starter-data-jpa`, which brings in Hibernate and the HikariCP connection pool.
Entity scanning	Finds `@Entity`, `@Embeddable` and `@MappedSuperclass` classes, so no `persistence.xml` is needed.
Derived query	A JPA query Spring Data creates automatically from a repository method's name.
@DataJpaTest	A test annotation that finds `@Entity` classes and sets up the Spring Data JPA repositories.
TestEntityManager	A test-specific alternative to the standard JPA `EntityManager`.

What the starter brings

The Data JPA starter adds Hibernate and HikariCP. Hibernate is the JPA implementation it includes. The `spring-boot-starter-jdbc` starter also pulls in HikariCP. `@SpringBootApplication` stands for `@SpringBootConfiguration`, `@EnableAutoConfiguration` and `@ComponentScan` together, so auto-configuration is on.

Choosing the database

Without `spring.datasource.url`, Boot tries an embedded database. Boot can set up embedded H2, HSQL and Derby on its own, with no connection URL. You can instead point at a JNDI `DataSource` with `spring.datasource.jndi-name`.

> **EXAM TRAP**
> Boot does not create JPA databases automatically for every database by default. That default covers only embedded H2, HSQL or Derby.

Finding entities and repositories

Entity scanning replaces `persistence.xml`. Boot finds entities by scanning, not from an XML list. Two annotations change where it looks, one for entities and one for repositories.

How Boot locates entities and repositories

Feature	Entities	Repositories
What is found	Classes with `@Entity`, `@Embeddable` or `@MappedSuperclass`	Interfaces that usually extend `Repository` or `CrudRepository`
Default search	Entity scanning, with no `persistence.xml`	The auto-configuration packages
Annotation to change the location	`@EntityScan`	`@EnableJpaRepositories`

Declaring a repository

A Spring Data JPA repository is an interface you declare. Spring Data creates JPA queries automatically from its method names. The example shows a paged `findAll` and a finder whose name spells out the query.

Example: Declares a city repository with a paged findAll method and a finder derived from its name.

```
public interface CityRepository
    extends Repository<City, Long> {
  Page<City> findAll(Pageable pageable);
  City findByNameAndStateAllIgnoringCase(
      String name, String state);
}
```

Queries beyond method names

Annotate a method with `@Query` for complex queries. Method names work well for simple finders, and `@Query` covers the rest. The JPA and MongoDB modules of Spring Data use one common repository infrastructure.

EXAM TRAP

Not every store builds finders from names the way JPA does. Cassandra repository support is more limited and needs `@Query` finder methods.

Startup and web view defaults

Repositories bootstrap in default, deferred or lazy mode. Set `spring.data.jpa.repositories.bootstrap-mode` to `deferred` or `lazy` to change the mode. In a web app, Boot registers `OpenEntityManagerInViewInterceptor` by default, allowing lazy loading in views. Setting `spring.jpa.open-in-view` to `false` stops it.

EXAM TRAP

Open EntityManager in View is already on by default in web apps. You turn it off, not on.

Testing the JPA layer

`@DataJpaTest` **scans entities and configures repositories.** It also sets up an embedded database when one is on the classpath. It logs SQL by default, and its `showSql` attribute turns that off. To include `@ConfigurationProperties` beans, add `@EnableConfigurationProperties`.

EXAM TRAP

`@DataJpaTest` does not scan regular `@Component` beans, even though it does scan `@Entity` classes.

Test transactions and helpers

Each data JPA test rolls back when it finishes. They are transactional by default. To turn that off for a whole class, use `Propagation.NOT_SUPPORTED`. Tests can inject a `TestEntityManager`, built for tests. Add `@AutoConfigureTestEntityManager` to get one in other Spring tests.

Example: Disables transaction management for every test in a data JPA test class.

```
@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
class MyNonTransactionalTests {
}
```

Key takeaways

- The Data JPA starter brings in Hibernate and the HikariCP connection pool.
- By default, Boot creates JPA databases automatically only for embedded H2, HSQL or Derby.
- Spring Data derives JPA queries from repository method names; use `@Query` for complex ones.
- `@EntityScan` moves entity scanning, and `@EnableJpaRepositories` moves repository scanning.
- By default, every data JPA test runs in a transaction that rolls back afterward.

Sources: SQL Databases :: Spring Boot (opens in a new tab) · Testing Spring Boot Applications :: Spring Boot (opens in a new tab) · Using the @SpringBootApplication Annotation :: Spring Boot (opens in a new tab) · Distributed Transactions With JTA :: Spring Boot (opens in a new tab) · Working with NoSQL Technologies :: Spring Boot (opens in a new tab)

Domain 3 | 10% of topics | 8 min read

Spring MVC

Overview

Every point below comes from Broadcom's official documentation; each section lists its sources.

Web Applications with Spring Boot

Start here

THINK FIRST

When a GET request reaches a Spring Boot web app, what decides which method runs and how its return value becomes the response?

Why it matters: Questions show a controller or a request and ask which annotation, mapping or return type produces the described response.

Spring Boot sets up a Spring MVC context when Spring MVC is on the classpath. Each request is matched to an annotated controller method. Its return value is then either written to the body by a message converter or resolved as a view.

Key terms

Term	Meaning
@RestController	A composed annotation combining `@Controller` and `@ResponseBody`, so every method writes to the response body.
HttpMessageConverter	The component that turns a return value into the body, for example JSON, and reads request bodies into objects.
HandlerInterceptor	A hook around handler execution with `preHandle`, `postHandle` and `afterCompletion` callbacks.
@PathVariable	Binds a captured URI template variable to a controller method argument.

Booting an MVC application

Spring MVC on the classpath gives you an MVC application context. The typical `main` method does nothing except call `SpringApplication.run(Class, String...)`. If Spring MVC and Spring WebFlux are both present, Spring MVC wins. A test of a reactive app then needs `spring.main.web-application-type` set.

Controllers found by scanning

`@Controller` **classes are detected through component scanning.** The stereotype registers the class as a bean and marks it as a web component. Annotated controllers need no base class or special interface, and their method signatures are flexible.

> **EXAM TRAP**
>
> Since Spring **6.0**, a type-level `@RequestMapping` on an interface alone no longer makes Spring MVC detect a controller when interface proxying is used.

Mapping a GET request

Use `@GetMapping` instead of a bare `@RequestMapping`. `@RequestMapping` can match by URL, HTTP method, parameters, headers and media types. Without a method it matches every HTTP method. `@GetMapping` also answers HEAD requests with no change to your method.

> **EXAM TRAP**
>
> Putting two `@RequestMapping` annotations on one method does not combine them. Spring logs a warning and uses only the first one.

Binding request data

Each kind of request data has its own binding annotation. URI variables are converted to the argument type automatically. A failed conversion raises `TypeMismatchException`. A simple value argument that nothing else resolves is treated like `@RequestParam`.

Where each annotation takes its value from

Annotation	Reads	Worth knowing
`@PathVariable`	A captured URI template variable	Converted to the argument type automatically
`@RequestParam`	Query parameters or form data	Required unless `required = false` or `Optional`
`@RequestBody`	The request body, via a message converter	Not reliable for form data; use `@RequestParam`

Writing the response

`@ResponseBody` **writes the return value; a plain name selects a view.** `ResponseEntity` behaves like `@ResponseBody` but also sets the status and headers. Its body is usually a value object rendered as JSON. In a REST controller, a `void` return can mean no body.

Return choices in a controller method

Return style	What reaches the client	Status and headers
Method with `@ResponseBody`	Return value serialized by a message converter	`ResponseEntity` adds status and headers
`ResponseEntity`	Body rendered by a registered converter	Set by the method
`@ResponseBody` on the class	Every method's return value, written to the body	`ResponseEntity` adds status and headers

Interceptors in the lifecycle

Every `HandlerMapping` supports interceptors around the handler. `preHandle` runs first; returning `false` skips the rest of the chain and the handler. `postHandle` runs after the handler. `afterCompletion` runs once the whole request has finished.

> **EXAM TRAP**
>
> `postHandle` cannot add a header for `@ResponseBody` or `ResponseEntity` methods. The response is already written and committed before it runs.

Validating a request body

Add `@Valid` or `@Validated` to a `@RequestBody` argument. Bean Validation then runs on the body. By default, failures raise `MethodArgumentNotValidException`, which becomes a **400** (BAD_REQUEST) response. An `Errors` or `BindingResult` argument lets the controller handle errors itself.

Running on a real port

`DEFINED_PORT` **starts an embedded server on a configured port.** With this `@SpringBootTest` web environment, the server listens on the port from `application.properties`, or on **8080** by default. For fast controller tests without a server, `@WebMvcTest` auto-configures MockMvc.

Key takeaways

- `@RestController` equals `@Controller` plus `@ResponseBody`, so no view is rendered.
- Prefer `@GetMapping`, because a bare `@RequestMapping` matches every HTTP method.
- `@RequestParam` arguments are required unless marked optional.
- Use `ResponseEntity` when the method must set the status or headers.
- Invalid `@RequestBody` input returns **400** by default.

Sources: Testing Spring Boot Applications :: Spring Boot (opens in a new tab) · Mapping Requests :: Spring Framework (opens in a new tab) · @ResponseBody :: Spring Framework (opens in a new tab) · Return Values :: Spring Framework (opens in a new tab) · Declaration :: Spring Framework (opens in a new tab) · ResponseEntity :: Spring Framework (opens in a new tab) · @RequestBody :: Spring Framework (opens in a new tab) · Interception :: Spring Framework (opens in a new tab)

REST Applications

Start here

THINK FIRST

How should a Spring Boot application build RestTemplate clients and report REST errors to the callers?

Why it matters: Questions describe a REST client or controller need and ask which Spring Boot class, bean or setting fits it.

Spring Boot gives you a RestTemplateBuilder, not a ready RestTemplate, because templates usually need customizing. Controllers can return RFC **9457** problem details for errors. Tests can call those endpoints with or without a running server.

Key terms

Term	Meaning
RestTemplate	The classic, imperative Spring class for calling remote REST services; it predates RestClient.
RestTemplateBuilder	An auto-configured builder that creates customized RestTemplate instances when you need them.
RestTemplateCustomizer	A bean holding an additive customization applied to every template the auto-configured builder builds.
ProblemDetail	An RFC **9457** error body; its status property sets the HTTP status of the response.
@WebMvcTest	A test annotation, often limited to a single controller, used with @MockitoBean to mock required collaborators.

Choosing a REST client

Pick an imperative or a reactive client to match the app. WebClient suits non-blocking reactive applications on Spring WebFlux. RestTemplate is the classic option and predates RestClient. Teams keep it for existing code they do not want to migrate, or because they know its API.

Spring's REST clients compared

Client	Style	When to pick it
WebClient	Non-blocking, reactive	Applications built on Spring WebFlux
RestClient	Imperative	Imperative code; inject the prototype RestClient.Builder
RestTemplate	Imperative, classic	Existing code you will not migrate, or a familiar API

No RestTemplate bean

Boot auto-configures RestTemplateBuilder, not a RestTemplate bean. Templates usually need customizing before use, so Boot gives you a builder instead. The builder applies sensible HttpMessageConverters and a suitable ClientHttpRequestFactory. Each builder method returns a new instance, so your changes affect only that use.

Example: Builds a RestTemplate with BASIC authentication from the injected builder.

```
RestTemplate myTemplate = builder
    .basicAuthentication("user", "password")
    .build();
```

- `basicAuthentication("user", "password")`: adds BASIC authentication to the template

EXAM TRAP

Autowiring a ready-made RestTemplate fails, because Boot provides no single auto-configured RestTemplate bean. Inject RestTemplateBuilder and build one.

Customizing every template

A RestTemplateCustomizer bean applies to all built templates. Every such bean is registered automatically with the auto-configured builder. A RestTemplateBuilder bean of your own replaces the auto-configured one. To pick a specific HTTP client factory instead of the auto-detected one, set `spring.http.clients.imperative.factory`.

EXAM TRAP

Your own RestTemplateBuilder bean does not get customizer beans applied automatically. Configure it with RestTemplateBuilderConfigurer if they should still apply.

Problem details for errors

Return ProblemDetail to send standard error details. Spring supports RFC **9457**, Problem Details for HTTP APIs. Any @ExceptionHandler or @RequestMapping method can return ProblemDetail or ErrorResponse. Its status property sets the HTTP status. For Spring MVC exceptions, extend ResponseEntityExceptionHandler and declare it as an @ControllerAdvice.

Reading errors on the client

Catch RestClientResponseException to decode an error body. With RestTemplate, the client catches RestClientResponseException. Its getResponseBodyAs methods decode the body into a target type, such as ProblemDetail. With WebClient, the matching exception is WebClientResponseException.

Testing REST endpoints

RestTestClient can test controllers with or without a server. RestTestClient wraps RestClient and adds a facade for verifying responses. bindToController uses mock requests and responses with no running server. bindToServer runs full end-to-end HTTP tests. Use expectAll(..) to assert every expectation even if one fails.

EXAM TRAP

@WebMvcTest does not scan regular @Component and @ConfigurationProperties beans. Provide mock collaborators with @MockitoBean instead.

TestRestTemplate setup

TestRestTemplate also needs the spring-boot-restclient dependency. It comes from the spring-boot-resttestclient module. Autowire it in a @SpringBootTest with a RANDOM_PORT web environment and @AutoConfigureTestRestTemplate. Adding spring-boot-restclient also enables RestClient.Builder auto-configuration, so put it on the main classpath if main code uses that builder.

Key takeaways

- Spring Boot auto-configures RestTemplateBuilder, not a RestTemplate bean; build templates from it.
- RestTemplateCustomizer beans apply to every template the auto-configured builder creates.
- Return ProblemDetail from a controller method to send an RFC **9457** error body.
- On the client, catch RestClientResponseException to read a RestTemplate error body.

Sources: Calling REST Services :: Spring Boot (opens in a new tab) · Error Responses :: Spring Framework (opens in a new tab) · RestTestClient :: Spring Framework (opens in a new tab) · Testing Spring Boot Applications :: Spring Boot (opens in a new tab)

Domain 4 | 10% of topics | 8 min read

Testing

Overview

Every point below comes from Broadcom's official documentation; each section lists its sources.

Testing Spring Applications

Start here

THINK FIRST

How do you give a Spring test the right beans, profiles and database state without slowing the suite down?

Why it matters: Questions describe a test setup and ask which annotation, profile or transaction behavior produces the stated result.

Spring tests load a real `ApplicationContext` without an application server and reuse it across the suite. Profiles decide which beans load, and each test's transaction rolls back unless you say otherwise.

Key terms

Term	Meaning
TestContext framework	Spring's test support that loads, caches and injects an application context, whatever test framework you use.
@ActiveProfiles	Test class annotation listing the bean definition profiles to activate for the test context.
Default profile	Profile whose beans load only when no other profile is specifically activated.
@Commit	Makes a test's transaction commit instead of rolling back.
@TestConfiguration	Nested configuration used in addition to the application's primary configuration.

Spring's test support

The TestContext framework works with JUnit, TestNG and others. It runs integration tests without an application server or other deployment environment. Once loaded, a test `ApplicationContext` is reused, so setup is paid once per suite. It can also inject beans into your test instances.

EXAM TRAP

The test context is not rebuilt before every test by default. It is loaded once and reused across the suite.

JUnit Jupiter composed annotations

`@SpringJUnitConfig` **bundles the extension and context setup.** It saves you from declaring two annotations on every test class. `@SpringJUnitWebConfig` adds web setup on top. `@EnabledIf` runs a test class or method only when its expression evaluates to true.

Two composed test annotations compared

Annotation	JUnit Jupiter part	Spring part
`@SpringJUnitConfig`	`@ExtendWith(SpringExtension.class)`	`@ContextConfiguration`
`@SpringJUnitWebConfig`	`@ExtendWith(SpringExtension.class)`	`@ContextConfiguration` and `@WebAppConfiguration`

Spring Boot web environments

The default `MOCK` **environment starts no embedded server.** `MOCK` loads a web `ApplicationContext` with a mock web environment. `DEFINED_PORT` starts a real server on the port from `application.properties`, or on **8080** by default.

EXAM TRAP

With `RANDOM_PORT` or `DEFINED_PORT`, client and server run in separate threads. A `@Transactional` test cannot roll back a transaction started on the server.

Test configuration and slices

A nested `@TestConfiguration` **adds to your main configuration.** A nested `@Configuration` class would replace the primary configuration instead. Several `@…Test` slice annotations in one test are not supported. Pick one and add the other slices' `@AutoConfigure…` annotations by hand.

Activating profiles in tests

`@ActiveProfiles` **picks the profiles for the test context.** List the profiles to activate when the test's `ApplicationContext` loads. Set `inheritProfiles` to stop inheriting a superclass's profiles. For logic, register a custom `ActiveProfilesResolver` through the `resolver` attribute.

EXAM TRAP

Setting `spring.profiles.active` as a JVM property or environment variable does not override `@ActiveProfiles`. The framework ignores that property when the annotation is present.

Transactions in tests

Each test's transaction rolls back by default. The framework creates a transaction for every test and rolls it back afterward. It uses a `PlatformTransactionManager` bean from the test's application context. Add `@Commit` when a test should keep its changes.

Preparing test data

`@Sql` **runs SQL scripts against a database in tests.** Put it on a test class or a test method. The `spring-jdbc` module can configure and launch an embedded database for these tests. `JdbcTestUtils` adds static JDBC helper methods for common database test cases.

JPA slice tests

`@DataJpaTest` **scans entities and configures repositories.** It also configures an embedded database when one is on the classpath. SQL queries are logged by default; turn this off with `showSql`. You can inject `TestEntityManager`, a test-focused alternative to the standard `EntityManager`.

Key takeaways

- A loaded test context is reused, so setup runs once per test suite.
- Test transactions roll back by default; use `@Commit` to keep the changes.
- `@ActiveProfiles` on a test class wins over the `spring.profiles.active` property.
- The default `MOCK` web environment does not start an embedded server.

Sources: Integration Testing :: Spring Framework (opens in a new tab) · Context Configuration with Environment Profiles :: Spring Framework (opens in a new tab) · Testing Spring Boot Applications :: Spring Boot (opens in a new tab) · Spring JUnit Jupiter Testing Annotations :: Spring Framework (opens in a new tab) · @ActiveProfiles :: Spring Framework (opens in a new tab) · @Sql :: Spring Framework (opens in a new tab) · JDBC Testing Support :: Spring Framework (opens in a new tab)

Advanced Testing with Spring Boot and MockMVC

Start here

THINK FIRST

When should you test a controller with standalone MockMvc instead of loading your full Spring MVC configuration?

Why it matters: Questions describe a web test goal and ask which MockMvc setup, test annotation or bean override fits it.

MockMvc runs full Spring MVC request handling through mock requests, with no running server. You choose how much configuration it loads, from one controller to your whole MVC setup.

Key terms

Term	Meaning
MockMvc	Runs Spring MVC request handling with mock Servlet objects instead of a running server.
Standalone setup	A MockMvc setup that points straight at chosen controllers and loads no Spring configuration.
@MockitoBean	Swaps a bean in the test ApplicationContext for a Mockito mock.
@SpringJUnitConfig	One annotation that combines the JUnit Jupiter SpringExtension with @ContextConfiguration.
@SpringBootTest	Spring Boot's annotation for full end-to-end integration tests.

Why plain unit tests fall short

Calling a controller directly skips most of Spring MVC. A plain unit test misses request mappings, data binding, message conversion, type conversion and validation. MockMvc fills that gap. It drives the `DispatcherServlet` with mock Servlet objects, so no server has to start.

EXAM TRAP

MockMvc does not start an embedded server. It replays full request handling with mock request and response objects.

Two ways to set up MockMvc

Load your MVC configuration, or point at controllers. The choice decides how much of your application each test covers. Loaded configuration is cached by the TestContext framework, which keeps a growing suite fast.

WebApplicationContext and standalone MockMvc setups compared

Feature	WebApplicationContext	Standalone
What it loads	Your actual Spring MVC configuration	No Spring configuration; one controller at a time
Dependencies	Override services with `@MockitoBean` or `@TestBean`	Inject mock dependencies into the controller by hand
Coverage	A more complete integration test	Needs extra integration tests for the MVC configuration

EXAM TRAP

A passing standalone test does not prove your Spring MVC configuration works. You still need integration tests for that.

What server-side tests can check

MockMvc sees inside request handling. Because these are server-side tests, you can assert which handler ran and whether a `HandlerExceptionResolver` handled an exception. You can also check the model and any binding errors. The `MockHttpServletResponse` keeps any forwarded or redirected URL, so you can assert it.

Surprises in the mock environment

Some servlet container behavior is missing. By default there is no context path and no jsessionid cookie. Forward, error and async dispatches do not happen, so JSPs are never rendered. To test with the full request URI, set `contextPath` and `servletPath` so mappings still match.

> **EXAM TRAP**
> Query parameters in the URI template get decoded, but values passed through `param(...)` are expected to be decoded already.

Writing expectations

`andExpectAll(..)` **reports every failed expectation.** Unlike `andExpect(..)`, it asserts all the expectations you pass and tracks each failure. Common expectations set on the instance always apply; dropping them takes a separate `MockMvc` instance. The `print()` handler writes result data to `System.out` unless an unhandled exception occurs.

Replacing beans with mocks

`@MockitoBean` **mocks a bean;** `@MockitoSpyBean` **spies on one.** Both override a bean in the test's ApplicationContext. On a field, the bean is found by the field's type. A `@Qualifier`, or else the field name, picks between several candidates.

How the two bean override annotations behave

Feature	@MockitoBean	@MockitoSpyBean
Strategy	REPLACE_OR_CREATE by default	WRAP: spies on an early instance of the real bean
When no bean exists	Creates a new bean	Not allowed: exactly one candidate bean is required
Special beans	A non-singleton bean becomes a singleton mock	A scoped proxy cannot be spied on

> **EXAM TRAP**
> With `enforceOverride = true`, the mock replaces the bean directly. It is never wrapped in an AOP proxy, even when the original bean uses `@Transactional`.

Keeping the test context cached

A loaded test context is reused by default. So the setup cost is paid only once per test suite. Qualifiers, including field names, decide whether a new context is needed. Name mock fields the same way across test classes to avoid extra contexts.

Composed test annotations

`@SpringJUnitConfig` **bundles the extension and context setup.** It combines `@ExtendWith(SpringExtension.class)` with `@ContextConfiguration`, and `@SpringJUnitWebConfig` adds `@WebAppConfiguration`. These annotations, and various Spring Boot Test annotations, often register the `SpringExtension` for you. For full end-to-end tests, Spring Boot offers `@SpringBootTest`.

Key takeaways

- Plain controller unit tests skip mappings, binding, conversion and validation; MockMvc covers them without a server.
- Standalone MockMvc tests need extra integration tests to verify your Spring MVC configuration.
- `@MockitoBean` replaces or creates a bean, while `@MockitoSpyBean` wraps exactly one existing bean.
- Test contexts are cached and reused by default, so keep mock field names consistent across classes.

Sources: Overview :: Spring Framework (opens in a new tab) · MockMvc :: Spring Framework (opens in a new tab) · Setup Options :: Spring Framework (opens in a new tab) · @MockitoBean and @MockitoSpyBean :: Spring Framework (opens in a new tab) · Spring JUnit Jupiter Testing Annotations :: Spring Framework (opens in a new tab) · MockMvc vs End-to-End Tests :: Spring Framework (opens in a new tab) · Performing Requests :: Spring Framework (opens in a new tab) · Defining Expectations :: Spring Framework (opens in a new tab)

Domain 5 | 16% of topics | 11 min read

Security

Overview

Every point below comes from Broadcom's official documentation; each section lists its sources.

Explain basic security concepts

Start here

THINK FIRST

When a request reaches a Spring Boot app with Spring Security, which pieces decide whether it gets through, and what do they do by default?

Why it matters: Questions describe a secured app and ask what Spring Security does by default or which component handles a step.

Spring Security plugs into the Servlet container as filters. A `DelegatingFilterProxy` hands each request to `FilterChainProxy`, which picks one `SecurityFilterChain`. Spring Boot adds a default chain that makes every request log in first.

Key terms

Term	Meaning
DelegatingFilterProxy	A filter registered with the Servlet container that passes all work to a Spring bean implementing `Filter`.
FilterChainProxy	The Spring Security filter bean that delegates to many filters through a `SecurityFilterChain`.
SecurityFilterChain	A set of security filters run in a fixed order; only the first chain matching a request runs.
AuthenticationManager	The component that authenticates a token such as `UsernamePasswordAuthenticationToken`.
UserDetailsService	A bean type that supplies users; Boot's default one holds a single user named `user` with a random password.

Security is built on filters

Spring Security's Servlet support rests on Servlet filters. A filter affects only the filters after it and the Servlet, so order matters a great deal. Authentication filters run before authorization filters. Because it uses a

standard filter, it protects any Servlet application, even one without Spring.

From container to Spring beans

`DelegatingFilterProxy` **links the container to Spring beans.** The container registers it like any filter, but it passes the work to a Spring bean. That bean is typically `FilterChainProxy`. It also clears the `SecurityContext` to avoid memory leaks and applies the `HttpFirewall`.

The three filter-level pieces compared

Feature	DelegatingFilterProxy	FilterChainProxy	SecurityFilterChain
What it is	A filter registered through standard Servlet container mechanisms	A special Spring Security filter that is a Spring bean	A group of security filters run in a set order
What it does	Passes every request on to a Spring bean of type `Filter`	Chooses the chain to use and delegates to its filters	Runs its filters, such as authentication before authorization

Choosing a filter chain

Only the first matching `SecurityFilterChain` **runs.** By default a chain matches every request; a `securityMatcher` overrides that. A chain bean with no `@Order` is considered last. If no chain matches a request, Spring Security leaves that request unprotected.

> **EXAM TRAP**
> Several matching chains do not all run. `FilterChainProxy` invokes only the first chain that matches the request.

What Spring Boot secures by default

With Spring Security on the classpath, web apps are secured. That includes Boot's `/error` endpoint. Auto-configuration adds `@EnableWebSecurity`, which publishes the default filter chain as a bean. That chain makes every request authenticate, by form login or HTTP Basic. Web requests go to the login page, while service requests get **401** Unauthorized.

The default user

Boot's default user is `user`**, with a random password.** The password is printed at WARN level when the app starts. Set `spring.security.user.password` to change the password, and `spring.security.user.name` for the name.

Switching the defaults off

A `SecurityFilterChain` **bean replaces Boot's default web security.** It does not disable the default `UserDetailsService` setup. To switch that off as well, add a `UserDetailsService`, `AuthenticationProvider` or `AuthenticationManager` bean.

EXAM TRAP

Adding your own `SecurityFilterChain` leaves the default `UserDetailsService` configuration in place.

HTTP Basic step by step

`BasicAuthenticationFilter` **turns credentials into a token.** It reads the username and password from the `HttpServletRequest` into a `UsernamePasswordAuthenticationToken`. The `AuthenticationManager` then authenticates that token. On failure, the `SecurityContextHolder` is cleared. The `BasicAuthenticationEntryPoint` sends a `WWW-Authenticate` header.

EXAM TRAP

HTTP Basic is on by default only until you provide servlet-based configuration. After that, you must configure HTTP Basic explicitly.

Built-in protections

CSRF protection is on by default for unsafe methods like POST. You write no extra code for it. A basic Java configuration also adds session fixation protection, a generated login form and logout. Java configuration arrived in Spring Security **3.2**, so no XML is needed.

EXAM TRAP

An unauthenticated request to a protected resource is not silently passed on. The `AuthorizationFilter` denies it by throwing an `AccessDeniedException`.

Key takeaways

- Spring Security works through Servlet filters, so the order of those filters matters.
- `FilterChainProxy` runs only the first `SecurityFilterChain` that matches a request.
- Boot's default chain makes every request authenticate, with form login and HTTP Basic.
- The default user is `user`, with a random password logged at startup.
- A `SecurityFilterChain` bean switches off Boot's default web security, not the default user setup.

Sources: Architecture :: Spring Security (opens in a new tab) · Java Configuration :: Spring Security (opens in a new tab) · Basic Authentication :: Spring Security (opens in a new tab) · Spring Security :: Spring Boot (opens in a new tab) · Servlet Applications :: Spring Security (opens in a new tab) · Hello Spring Security :: Spring Security (opens in a new tab) · Cross Site Request Forgery (CSRF) :: Spring Security (opens in a new tab)

Use Spring Security to configure Authentication and Authorization

Start here

THINK FIRST

When a request reaches your app, which Spring Security parts decide who the user is and what they may do?

Why it matters: Questions trace a request through authentication and authorization and ask which component acts, decides or fails.

Authentication proves who the user is, and the result is kept in the `SecurityContextHolder`. Authorization then checks that user's granted authorities, and an `AuthorizationManager` makes the final access decision.

Key terms

Term	Meaning
SecurityContextHolder	Where Spring Security keeps the details of the currently authenticated user.
ProviderManager	The usual `AuthenticationManager`; it delegates to a list of `AuthenticationProvider` instances.
GrantedAuthority	A permission held by the user, usually application-wide rather than tied to one domain object.
AuthenticationEntryPoint	The component that asks an unauthenticated client for credentials.

Where the user is stored

The `SecurityContextHolder` holds the authenticated user. If it contains a value, Spring Security treats it as the current user, however it got there. By default it uses a `ThreadLocal`, so methods on the same thread can read it without receiving it as an argument.

EXAM TRAP

Spawned threads do not share the identity by default. Use the `MODE_INHERITABLETHREADLOCAL` strategy when child threads must act as the same user.

The Authentication object

An `Authentication` carries credentials into the manager. As input to the `AuthenticationManager`, its `isAuthenticated()` method returns `false`. Credentials are often a password. After a successful login, `ProviderManager` clears sensitive credentials from the returned object by default, so they are not leaked.

How ProviderManager works

`ProviderManager` **delegates to a list of providers.** Each `AuthenticationProvider` supports one kind of authentication. `DaoAuthenticationProvider` handles a username and password, while

`JwtAuthenticationProvider` authenticates a JWT token. If no provider can authenticate the request, it fails with a `ProviderNotFoundException`.

EXAM TRAP

A `ProviderNotFoundException` is not a wrong-password error. It means no configured provider supports that type of `Authentication`.

The default username and password setup

A simple user setup wires several pieces for you. It registers an in-memory `UserDetailsService` and adds `DaoAuthenticationProvider` to the default `AuthenticationManager`. It also turns on HTTP Basic and Form Login. `InMemoryUserDetailsManager` implements `UserDetailsService` for users kept in memory.

EXAM TRAP

`User#withDefaultPasswordEncoder` belongs in sample applications only. It is considered unsafe for production.

Roles and authorities

Role rules add the `ROLE_` prefix by default. A rule that requires role `USER` looks for the authority `ROLE_USER`. `AuthorizationManager` instances make the final decision for request-based, method-based and message-based authorization.

EXAM TRAP

Granting an authority named `USER` does not satisfy a rule for role `USER`. By default that rule checks for `ROLE_USER`.

When access fails

A missing login and a denied call end differently. An unauthenticated client meets an `AuthenticationEntryPoint`, which asks for credentials. When method authorization fails, an `AccessDeniedException` is thrown and the client gets a **403**.

Not authenticated versus denied by method authorization

Feature	Not authenticated	Method access denied
Component that acts	`AuthenticationEntryPoint`	Method interceptor, then `ExceptionTranslationFilter`
Typical response	Redirect to a login page or a `WWW-Authenticate` header	A **403** status code
Event or exception	Credentials are requested	`AuthorizationDeniedEvent` and `AccessDeniedException`

Turning on method security

Spring Boot leaves method authorization off by default. Add `@EnableMethodSecurity` to any `@Configuration` class to turn it on. It enables `@PreAuthorize`, `@PostAuthorize`, `@PreFilter` and `@PostFilter`. Unannotated methods stay unsecured, so also declare a catch-all rule in `HttpSecurity`.

EXAM TRAP

`@EnableGlobalMethodSecurity` is deprecated. Migrate to `@EnableMethodSecurity` instead.

Key takeaways

- `ProviderManager` delegates to providers and throws `ProviderNotFoundException` when none supports the request.
- The `SecurityContextHolder` uses a `ThreadLocal` by default, so the user is bound to the current thread.
- Role rules check for the `ROLE_` prefix by default, so role `USER` means authority `ROLE_USER`.
- Method security stays off in Spring Boot until you add `@EnableMethodSecurity`.

Sources: Servlet Authentication Architecture :: Spring Security (opens in a new tab) · Authorization Architecture :: Spring Security (opens in a new tab) · Username/Password Authentication :: Spring Security (opens in a new tab) · In-Memory Authentication :: Spring Security (opens in a new tab) · Method Security :: Spring Security (opens in a new tab)

Define Method-level Security

Start here

THINK FIRST

How would you secure a Spring service method, and what does the caller see when the check fails?

Why it matters: Questions show annotated service methods and ask whether they are secured, which annotation applies and what the caller gets back.

Method security stays off until you switch it on with `@EnableMethodSecurity`. It is built on Spring AOP, so annotated methods are checked when they are called. A failed check ends in a **403** response.

Key terms

Term	Meaning
@EnableMethodSecurity	Annotation on a `@Configuration` class that turns on method-level authorization.
@PreAuthorize	Recommended annotation for authorizing method calls; it supersedes the legacy `@Secured` annotation.
returnObject	In a `@PostAuthorize` expression, the value the method is about to return.

Term	Meaning
filterObject	In a `@PreFilter` expression, the collection element being tested.
hasRole	Expression shortcut for `hasAuthority` that adds the `ROLE_` prefix, or the configured default prefix.

Turning method security on

Method security stays off until you enable it. Spring Boot Starter Security does not activate it for you. Annotate any `@Configuration` class with `@EnableMethodSecurity`, or add the `method-security` element to an XML configuration file. The older `@EnableGlobalMethodSecurity` and the `global-method-security` element are deprecated.

EXAM TRAP

Adding the Spring Boot security starter alone does not turn on method-level authorization. You still have to enable it explicitly.

What @EnableMethodSecurity enables

Four pre and post annotations work by default. `@EnableMethodSecurity` enables `@PreAuthorize`, `@PostAuthorize`, `@PreFilter` and `@PostFilter`. For `@RolesAllowed`, `@PermitAll` and `@DenyAll`, set `jsr250Enabled = true`. `@Secured` is a legacy option, and `@PreAuthorize` is the recommended replacement.

EXAM TRAP

`@Secured` still exists, but it is the legacy choice. When a question asks for the recommended annotation, pick `@PreAuthorize`.

Checking results and filtering arguments

Some annotations test the data, not only the caller. `@PostAuthorize` lets a method return only when its expression passes for the returned value. `@PreFilter` removes each collection element whose expression fails. In role checks, `hasRole` adds the `ROLE_` prefix for you.

Two data-aware annotations compared

Annotation	What it tests	Expression variable
`@PostAuthorize`	The value the method returns	`returnObject`
`@PreFilter`	Each element of a collection argument	`filterObject`

Combining annotations

Every annotation on a method must pass. Different annotations on one method are checked one at a time, and the call is authorized only if all of them pass. You cannot repeat the same annotation on one method. A method-level annotation overrides one declared on its class.

EXAM TRAP

Placing `@PreAuthorize` twice on the same method is not supported, even though different annotations can be combined.

Unannotated methods and failed checks

Methods without an annotation are not secured. Declare a catch-all authorization rule in `HttpSecurity` to cover them. When a check fails, the interceptor throws an `AccessDeniedException`. The `ExceptionTranslationFilter` catches it and returns a **403** status code.

XML element defaults

The newer XML element turns pre and post annotations on by default. The older `global-method-security` element is deprecated, and `method-security` replaces it. Keeping the old one means adding the `spring-security-access` dependency to your build.

The two XML elements compared

Setting	`method-security`	`global-method-security`
Pre and post annotations	`pre-post-enabled`, defaults to true	`pre-post-annotations`, defaults to disabled
`@Secured` support	`secured-enabled`, defaults to false	`secured-annotations`, defaults to disabled
Status	Current element	Deprecated; needs `spring-security-access`

AspectJ mode

AspectJ ignores security annotations on interfaces. Setting `mode` to `aspectj` makes Spring Security intercept calls with AspectJ. Java does not inherit interface annotations, so place them on the class. Secured methods must be woven with `AnnotationSecurityAspect` from `spring-security-aspects`.

EXAM TRAP

With AspectJ, annotating only the interface leaves those methods unsecured. Put the security annotation on the class.

Key takeaways

- Method security is off by default; enable it with `@EnableMethodSecurity` on a `@Configuration` class.
- Unannotated methods are not secured, so add a catch-all rule in `HttpSecurity`.
- A failed check throws `AccessDeniedException`, which becomes a **403** response.
- Prefer `@PreAuthorize` over the legacy `@Secured` annotation.

Sources: Method Security :: Spring Security (opens in a new tab) · Method Security :: Spring Security (opens in

a new tab)

Domain 6 | 16% of topics | 9 min read

Spring Boot

Overview

Every point below comes from Broadcom's official documentation; each section lists its sources.

Spring Boot Feature Introduction

Start here

THINK FIRST

How does Spring Boot decide which value wins when the same property is set in several places?

Why it matters: Questions describe a configuration setup and ask which property value, profile or file the application ends up using.

Spring Boot reads properties from many sources in a fixed order. Later sources override earlier ones, so the order decides the winning value. Profiles add another layer of override on top.

Key terms

Term	Meaning
PropertySource order	The fixed order in which Spring Boot reads property sources; later ones override earlier ones.
Profile	A name you list in `spring.profiles.active` to make it active; its profile-specific files override the non-specific ones.
Default profile	The profile named `default` that Spring Boot enables when no other profile is active.
Property placeholder	A `${...}` reference to a property that can carry a default value after a colon.

Override order

Later property sources override earlier ones. Spring Boot reads its property sources in a set order built for sensible overriding. A value passed on the command line wins over the same key in any configuration file.

EXAM TRAP

A value in `application.properties` does not beat a command line argument. Values from the command line always outrank values from files.

Properties versus YAML

In the same location, `.properties` beats YAML. YAML works whenever SnakeYAML is on the classpath, and `spring-boot-starter` brings it in when you use starters. You cannot load YAML files with `@PropertySource` or `@TestPropertySource`.

> **EXAM TRAP**
> Adding a YAML file beside a `.properties` file with the same key does not change the value. The `.properties` file wins.

Activating profiles

`spring.profiles.active` **sets the active profiles.** With nothing active, Spring Boot enables a profile named `default`, which `spring.profiles.default` can rename. To add profiles without replacing the active ones, use `spring.profiles.include`.

The three profile properties compared

Feature	`spring.profiles.active`	`spring.profiles.default`	`spring.profiles.include`
Purpose	Names the active profiles	Names the fallback profile	Adds more active profiles
Used when	You choose which profiles run	No profile is active	You extend the active list instead of replacing it

Profile-specific files

Profile-specific files override the non-specific ones. They load from the same locations as `application.properties`. When several profiles are active, the last one wins. Keep `spring.profiles.active` and `spring.profiles.default` in non-profile-specific documents.

> **EXAM TRAP**
> Setting `spring.profiles.active` inside a profile-specific document is not allowed. It belongs only in non-profile-specific documents.

Config file names and locations

`spring.config.name` **renames the `application` file.** `spring.config.location` replaces the default locations instead of adding to them. By default, a named location that does not exist throws `ConfigDataLocationNotFoundException`, and the application does not start.

> **EXAM TRAP**
> Pointing `spring.config.location` at a folder does not add it to the defaults. The locations you set replace the default ones.

Placeholders with defaults

A colon in a placeholder sets its default value. Write the property name, then a colon, then the default, as in `${name:default}`.

Key takeaways

- Later property sources override earlier ones, and command line properties always beat files.
- In one location, a `.properties` file wins over a YAML file.
- `spring.profiles.include` adds profiles; it never replaces the ones `spring.profiles.active` sets.
- By default, a missing config location stops startup with `ConfigDataLocationNotFoundException`.

Sources: Externalized Configuration :: Spring Boot (opens in a new tab) · Profiles :: Spring Boot (opens in a new tab)

Spring Boot Properties and Autoconfiguration

Start here

THINK FIRST

How does Spring Boot decide what to configure for you, and how do you take back control when its choice is wrong?

Why it matters: Questions show a Boot application and ask what auto-configuration applies, how to exclude it, or how your own beans override it.

Auto-configuration looks at the jars on your classpath and configures beans to match. Its classes are standard `@Configuration` classes. Define your own bean and the default backs away; exclude a class when you do not want it at all.

Key terms

Term	Meaning
Auto-configuration	Boot configuring your application from the jar dependencies you have added.
@AutoConfiguration	The annotation on auto-configuration classes; it is meta-annotated with `@Configuration`.
@ConditionalOnMissingBean	A condition commonly placed on auto-configuration so you can override its defaults when they do not suit you.
Conditions report	A log of which auto-configuration applied and why, printed with `--debug`.

Opting in

Add `@SpringBootApplication` or `@EnableAutoConfiguration` once. Put one of them on a `@Configuration` class, ideally your primary one. Add only one such annotation in the whole application.

EXAM TRAP

Adding `@EnableAutoConfiguration` to several configuration classes is not recommended. Use exactly one, preferably on the primary class.

Classpath-driven defaults

Boot configures beans from the jars you have added. For example, with HSQLDB on the classpath and no connection beans of your own, Boot sets up an in-memory database. A jar lists its candidates, one class name per line, in `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`.

Overriding a default

Define your own bean and that part of auto-configuration backs away. Add your own `DataSource` bean, and the embedded database support steps aside. This works because defaults are commonly guarded with `@ConditionalOnMissingBean`.

EXAM TRAP

You do not need to exclude an auto-configuration just to replace one bean. Defining your own bean is enough for the default to back away.

Excluding auto-configuration

Use the `exclude` attribute to switch off unwanted classes. When the class is not on the classpath, use `excludeName` with its fully qualified name. The `spring.autoconfigure.exclude` property controls the same list from configuration.

Three ways to exclude an auto-configuration class

Option	Where you set it	When to use it
`exclude`	On `@SpringBootApplication`	The class is on the classpath
`excludeName`	On the annotation, as a fully qualified name	The class is not on the classpath
`spring.autoconfigure.exclude`	As a property	You prefer to control the list by property

Conditions and ordering

Use bean conditions only on auto-configuration classes. Those classes load after user-defined bean definitions, so `@ConditionalOnBean` and `@ConditionalOnMissingBean` see your beans. `@ConditionalOnProperty` checks an Environment property named by `prefix` and `name`. Order classes with `before` and `after`, or `@AutoConfigureBefore` and `@AutoConfigureAfter`.

Seeing what applied

Start with `--debug` to log the conditions report. The report shows which auto-configuration is applied and why. If you write a starter with configuration keys, give them a unique namespace, never `server`, `management` or `spring`.

Key takeaways

- Auto-configuration reacts to the jar dependencies on your classpath.
- Your own bean, such as a `DataSource`, makes the matching default back away.
- Exclude classes with `exclude`, `excludeName` or `spring.autoconfigure.exclude`.
- Run with `--debug` to see the conditions report.

Sources: Auto-configuration :: Spring Boot (opens in a new tab) · Creating Your Own Auto-configuration :: Spring Boot (opens in a new tab)

Spring Boot Actuator

Start here

THINK FIRST

How would you decide which Actuator endpoints to expose, who may call them, and what custom data they report?

Why it matters: Questions describe an Actuator setup and ask what is exposed, who can reach it, or how to add custom metrics and health checks.

Out of the box, only `health` is exposed over HTTP and JMX, and access to every endpoint except `shutdown` and `heapdump` is unrestricted. With Spring Security on the classpath and no `SecurityFilterChain` bean of your own, every actuator except `/health` is secured. You extend Actuator with your own meters and health indicators.

Key terms

Term	Meaning
Exposure	Whether an endpoint is published over HTTP or JMX, set with include and exclude lists.
Base path	The URL prefix for management endpoints, `/actuator` by default.
MeterRegistry	The Spring-managed registry you inject to register custom metrics.
HealthIndicator	An interface a bean implements to add its own health information.

Default exposure

Only `health` is exposed by default. This applies over both HTTP and JMX. Over HTTP, the endpoint ID follows the `/actuator` prefix, so health answers at `/actuator/health`. Access is a separate setting:

every endpoint except `shutdown` and `heapdump` is unrestricted by default.

EXAM TRAP

Unrestricted access is not the same as exposure. An endpoint can allow access and still not be published over HTTP until you include it.

Include and exclude

`exclude` **wins over** `include`**.** The include list names endpoints to expose, and the exclude list names endpoints to hide. Use `*` to select every endpoint. If an ID appears in both lists, it stays hidden.

Settings that shape how endpoints are reached

Setting	What it does	Example use
`include`	Lists endpoint IDs to expose	`*` to expose every endpoint
`exclude`	Lists endpoint IDs to hide; beats include	Hide one ID that `*` selected
`management.endpoints.web.base-path`	Changes the management URL prefix	Replace the `/actuator` prefix
Per-endpoint `access` property	Sets permitted access for that endpoint	Limit access to a single endpoint

EXAM TRAP

Putting an ID in both lists does not expose it. The exclude list takes precedence.

Turning off HTTP

Set the management port to -1 to stop HTTP exposure. Use this when you want no endpoints reachable over HTTP at all. Changing the base path only moves the endpoints to a new prefix; it does not hide them.

Default security

With Spring Security present, every actuator except `/health` **is secured.** This holds while no other `SecurityFilterChain` bean exists. Define your own `SecurityFilterChain` and auto-configuration backs off, leaving the rules to you. Inside it, `EndpointRequest.toAnyEndpoint()` matches a request to any endpoint.

CSRF and write operations

POST, PUT and DELETE endpoints get 403 by default. Spring Boot keeps Spring Security's defaults, so CSRF protection is on. That blocks write calls such as `shutdown` and `loggers` under the default configuration.

EXAM TRAP

A **403** from `shutdown` under default security is CSRF protection at work. It is not a sign that the endpoint is missing.

Health roles

With no roles set, any authenticated user is authorized. That is the default for the health endpoint. To restrict who sees it, set `management.endpoint.health.roles`.

Custom metrics

Inject the Spring-managed `MeterRegistry` to add metrics. Avoid the static methods on `Metrics`. They use a global registry that Spring does not manage. When your metrics depend on other beans, register them with a `MeterBinder` instead.

> **EXAM TRAP**
>
> Static `Metrics` methods compile and run, but they register meters in a global registry outside Spring's control.

Custom health indicators

Register a bean that implements `HealthIndicator`. Implement its `health()` method and return a `Health` response. The response should carry a status and may add extra details for display.

Key takeaways

- Only `health` is exposed by default, over HTTP and JMX alike.
- In exposure settings, `exclude` beats `include`, and `*` selects every endpoint.
- Defining your own `SecurityFilterChain` makes Spring Boot's actuator security back off.
- Register meters on the injected `MeterRegistry`, or use a `MeterBinder` when they need other beans.

Sources: Endpoints :: Spring Boot (opens in a new tab) · Metrics :: Spring Boot (opens in a new tab) · Monitoring and Management Over HTTP :: Spring Boot (opens in a new tab)

MockCert

Take a Spring Certified Professional mock exam the way the real one feels

5 timed mock exams · 60 questions in 130 minutes each · one is free

- Free to start: study notes, explained quizzes, flashcards and the first mock exam
- Timed mock exams built to the official exam guide
- A verifiable Certificate of Assessment when you pass a premium mock

Explore Spring Certified Professional on MockCert



Open it on your phone

mockcert.com/spring-professional/

Scan the code with your phone's camera.