

A Guide to Spring Framework Annotations

by John Thompson 🐉 MVB · Sep. 24, 17 · Java Zone

Build vs Buy a Data Quality Solution: Which is Best for You? Gain insights on a hybrid approach. Download white paper now!

The Java programming language provided support for annotations from Java 5.0 onward. Leading Java frameworks were quick to adopt annotations, and the Spring Framework started using annotations from the 2.5 release. Due to the way they are defined, annotations provide a lot of context in their declaration.

Prior to annotations, the behavior of the Spring Framework was largely controlled through XML configuration. Today, the use of annotations provide us tremendous capabilities in how we configure the behaviors of the Spring Framework.

In this post, we'll take a look at the annotations available in the Spring Framework.

Core Spring Framework Annotations

@Required

This annotation is applied to bean setter methods. Consider a scenario where you need to enforce a required property. The `@Required` annotation indicates that the affected bean must be populated at configuration time with the required property. Otherwise, an exception of type `BeanInitializationException` is thrown.

@Autowired

This annotation is applied to fields, setter methods, and constructors. The `@Autowired` annotation injects object dependency implicitly.

When you use `@Autowired` on fields and pass the values for the fields using the property name, Spring will automatically assign the fields with the passed values.

You can even use `@Autowired` on private properties, as shown below. (This is a very poor practice though!)

```
1 public class Customer {  
2     @Autowired
```

```
3     private Person person;
4     private int type;
5 }
```

When you use `@Autowired` on setter methods, Spring tries to perform it by Type autowiring on the method. You are instructing Spring that it should initiate this property using a setter method where you can add your custom code, like initializing any other property with this property.

```
public class Customer {
1
2     private Person person;
        @Autowired
3
4     public void setPerson (Person person) {
5         this.person=person;
6     }
7 }
```

Consider a scenario where you need an instance of class A, but you do not store A in the field of the class. You just use A to obtain an instance of B, and you are storing B in this field. In this case, setter method autowiring will better suit you. You will not have class-level unused fields.

When you use `@Autowired` on a constructor, then constructor injection happens at the time of object creation. It tells the constructor to autowire when used as a bean. One thing to note here is that only one constructor of any bean class can carry the `@Autowired` annotation.

```
1 @Component
2 public class Customer {
3     private Person person;
4     @Autowired
5     public Customer (Person person) {
6         this.person=person;
7     }
8 }
```

NOTE: As of Spring 4.3, `@Autowired` became optional on classes with a single constructor. In the above example, Spring would still inject an instance of the Person class if you omitted the `@Autowired` annotation.

@Qualifier

This annotation is used along with the @Autowired annotation. When you need more control of the dependency injection process, @Qualifier can be used. @Qualifier can be specified on individual constructor arguments or method parameters. This annotation is used to avoid the confusion that occurs when you create more than one bean of the same type and want to wire only one of them with a property.

Consider an example where an interface BeanInterface is implemented by two beans, BeanB1 and BeanB2.

```
1  @Component
2  public class BeanB1 implements BeanInterface {
3      //
4  }
5  @Component
6  public class BeanB2 implements BeanInterface {
7      //
8  }
```

Now if BeanA autowires this interface, Spring will not know which one of the two implementations to inject.

One solution to this problem is the use of the @Qualifier annotation.

```
1  @Component
2  public class BeanA {
3      @Autowired
4      @Qualifier("beanB2")
5      private IBean dependency;
6      ...
7  }
```

With the @Qualifier annotation added, Spring will now know which bean to autowire, where beanB2 is the name of BeanB2.

@Configuration

This annotation is used on classes that define beans. @Configuration is an analog for an XML configuration file – it is configuration using Java classes. A Java class annotated with @Configuration is a configuration by itself and will have methods to instantiate and configure the dependencies.

Here is an example:

```
1  @Configuration
2  public class DataConfig {
3      @Bean
4      public DataSource source() {
5          DataSource source = new OracleDataSource();
6          source.setURL();
7          source.setUser();
8          return source;
9      }
10     @Bean
11     public PlatformTransactionManager manager() {
12         PlatformTransactionManager manager = new BasicDataSourceTransactionMa
13 nager();
14         manager.setDataSource(source());
15         return manager;
16     }
```

@ComponentScan

This annotation is used with the @Configuration annotation to allow Spring to know the packages to scan for annotated components. @ComponentScan is also used to specify base packages using basePackageClasses or basePackage attributes to scan. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation.

@Bean

This annotation is used at the method level. The @Bean annotation works with @Configuration to create Spring beans. As mentioned earlier, @Configuration will have methods to instantiate and configure dependencies. Such methods will be annotated with @Bean. The method annotated with this annotation works as the bean ID, and it creates and returns the actual bean.

Here is an example:

```
1  @Configuration
2  public class AppConfig {
3      @Bean
4      public Person person() {
5          return new Person(address());
6      }
7      @Bean
8      public Address address() {
9          return new Address();
10     }
11 }
```

@Lazy

This annotation is used on component classes. By default, all autowired dependencies are created and configured at startup. But if you want to initialize a bean lazily, you can use the `@Lazy` annotation over the class. This means that the bean will be created and initialized only when it is first requested for. You can also use this annotation on `@Configuration` classes. This indicates that all `@Bean` methods within that `@Configuration` should be lazily initialized.

@Value

This annotation is used at the field, constructor parameter, and method parameter levels. The `@Value` annotation indicates a default value expression for the field or parameter to initialize the property with. As the `@Autowired` annotation tells Spring to inject an object into another when it loads your application context, you can also use the `@Value` annotation to inject values from a property file into a bean's attribute. It supports both `#{...}` and `${...}` placeholders.

Spring Framework Stereotype Annotations

@Component

This annotation is used on classes to indicate a Spring component. The `@Component` annotation marks the Java class as a bean or component so that the component-scanning mechanism of Spring can add it into the application context.

@Controller

The `@Controller` annotation is used to indicate the class is a Spring controller. This annotation

can be used to identify controllers for Spring MVC or Spring WebFlux.

@Service

This annotation is used on a class. @Service marks a Java class that performs some service, such as executing business logic, performing calculations, and calling external APIs. This annotation is a specialized form of the @Component annotation intended to be used in the service layer.

@Repository

This annotation is used on Java classes that directly access the database. The @Repository annotation works as a marker for any class that fulfills the role of repository or Data Access Object.

This annotation has an automatic translation feature. For example, when an exception occurs in the @Repository, there is a handler for that exception and there is no need to add a try-catch block.

Spring Boot Annotations

@EnableAutoConfiguration

This annotation is usually placed on the main application class. The @EnableAutoConfiguration annotation implicitly defines a base “search package”. This annotation tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

@SpringBootApplication

This annotation is used on the application class while setting up a Spring Boot project. The class that is annotated with the @SpringBootApplication must be kept in the base package. The one thing that the @SpringBootApplication does is a component scan. But it will scan only its sub-packages. As an example, if you put the class annotated with @SpringBootApplication in com.example, then @SpringBootApplication will scan all its sub-packages, such as com.example.a, com.example.b, and com.example.a.x.

The @SpringBootApplication is a convenient annotation that adds all the following:

- @Configuration
- @EnableAutoConfiguration
- @ComponentScan

Spring MVC and REST Annotations

@Controller

This annotation is used on Java classes that play the role of controller in your application. The `@Controller` annotation allows autodetection of component classes in the classpath and auto-registering bean definitions for them. To enable autodetection of such annotated controllers, you can add component scanning to your configuration. The Java class annotated with `@Controller` is capable of handling multiple request mappings.

This annotation can be used with Spring MVC and Spring WebFlux.

@RequestMapping

This annotation is used at both the class and method level. The `@RequestMapping` annotation is used to map web requests onto specific handler classes and handler methods. When `@RequestMapping` is used on the class level, it creates a base URI for which the controller will be used. When this annotation is used on methods, it will give you the URI on which the handler methods will be executed. From this, you can infer that the class level request mapping will remain the same whereas each handler method will have their own request mapping.

Sometimes you may want to perform different operations based on the HTTP method used, even though the request URI may remain the same. In such situations, you can use the `method` attribute of `@RequestMapping` with an HTTP method value to narrow down the HTTP methods in order to invoke the methods of your class.

Here is a basic example of how a controller along with request mappings work:

```
1  @Controller
2  @RequestMapping("/welcome")
3  public class WelcomeController {
4      @RequestMapping(method = RequestMethod.GET)
5      public String welcomeAll() {
6          return "welcome all";
7      }
8  }
```

In this example, only GET requests to `/welcome` is handled by the `welcomeAll()` method.

This annotation also can be used with Spring MVC and Spring WebFlux.

The `@RequestMapping` annotation is very versatile. Please see my in-depth post on Request Mapping [here](#).

@CookieValue

This annotation is used at method parameter level. `@CookieValue` is used as an argument of a request mapping method. The HTTP cookie is bound to the `@CookieValue` parameter for a given cookie name. This annotation is used in the method annotated with `@RequestMapping`.

Let us consider that the following cookie value is received with an HTTP request:

```
JSESSIONID=418AB76CD83EF94U85YD34W
```

To get the value of the cookie, use `@CookieValue` like this:

```
1 @RequestMapping("/cookieValue")
2     public void getCookieValue(@CookieValue "JSESSIONID" String cookie){
3 }
```

@CrossOrigin

This annotation is used both at the class and method levels to enable cross-origin requests. In many cases, the host that serves JavaScript will be different from the host that serves the data. In such a case, Cross Origin Resource Sharing (CORS) enables cross-domain communication. To enable this communication, you just need to add the `@CrossOrigin` annotation.

By default, the `@CrossOrigin` annotation allows all origin, all headers, the HTTP methods specified in the `@RequestMapping` annotation, and a `maxAge` of 30 min. You can customize the behavior by specifying the corresponding attribute values.

An example of using `@CrossOrigin` at both the controller and handler method levels is below:

```
1 @CrossOrigin(maxAge = 3600)
2 @RestController
3 @RequestMapping("/account")
4 public class AccountController {
5
6     @CrossOrigin(origins = "http://example.com")
7     @RequestMapping("/message")
8     public Message getMessage() {
9         // ...
10 }
```



```
11
12     @RequestMapping("/note")
13     public Note getNote() {
14         // ...
15     }
16 }
```

In this example, both the `getExample()` and `getNote()` methods will have a `maxAge` of 3600 seconds. Also, `getExample()` will only allow cross-origin requests from `http://example.com`, while `getNote()` will allow cross-origin requests from all hosts.

Composed @RequestMapping Variants

Spring framework 4.3 introduced the following method-level variants of `@RequestMapping` annotation to better express the semantics of the annotated methods. Using these annotations has become the standard ways of defining the endpoints. They act as wrappers to `@RequestMapping`.

These annotations can be used with Spring MVC and Spring WebFlux.

@GetMapping

This annotation is used for mapping HTTP GET requests onto specific handler methods.

`@GetMapping` is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.GET)`.

@PostMapping

This annotation is used for mapping HTTP POST requests onto specific handler methods.

`@PostMapping` is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.POST)`.

@PutMapping

This annotation is used for mapping HTTP PUT requests onto specific handler methods.

`@PutMapping` is a composed annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.PUT)`.

@PatchMapping

This annotation is used for mapping HTTP PATCH requests onto specific handler methods. `@PatchMapping` is a composed annotation that acts as a shortcut for `@RequestMapping` (method = RequestMethod.PATCH).

@DeleteMapping

This annotation is used for mapping HTTP DELETE requests onto specific handler methods. `@DeleteMapping` is a composed annotation that acts as a shortcut for `@RequestMapping` (method = RequestMethod.DELETE).

@ExceptionHandler

This annotation is used at method levels to handle exceptions at the controller level. The `@ExceptionHandler` annotation is used to define the class of exception it will catch. You can use this annotation on methods that should be invoked to handle an exception. The `@ExceptionHandler` values can be set to an array of Exception types. If an exception is thrown that matches one of the types in the list, then the method annotated with the matching `@ExceptionHandler` will be invoked.

@InitBinder

This annotation is a method-level annotation that plays the role of identifying the methods that initialize the `WebDataBinder` — a `DataBinder` that binds the request parameter to JavaBean objects. To customize request parameter data binding, you can use `@InitBinder` annotated methods within our controller. The methods annotated with `@InitBinder` includes all argument types that handler methods support.

The `@InitBinder` annotated methods will get called for each HTTP request if you don't specify the value element of this annotation. The value element can be a single or multiple form names or request parameters that the init binder method is applied to.

@Mappings and @Mapping

This annotation is used on fields. The `@Mapping` annotation is a meta-annotation that indicates a web mapping annotation. When mapping different field names, you need to configure the source field to its target field, and to do that, you have to add the `@Mappings` annotation. This annotation accepts an array of `@Mapping` having the source and the target fields.

@MatrixVariable

This annotation is used to annotate request handler method arguments so that Spring can inject the relevant bits of a matrix URI. Matrix variables can appear on any segment each separated by a semicolon. If a URL contains matrix variables, the request mapping pattern must represent them with a URI template. The `@MatrixVariable` annotation ensures that the request is matched with the correct matrix variables of the URI.

@PathVariable

This annotation is used to annotate request handler method arguments. The `@RequestMapping` annotation can be used to handle dynamic changes in the URI where a certain URI value acts as a parameter. You can specify this parameter using a regular expression. The `@PathVariable` annotation can be used declare this parameter.

@RequestAttribute

This annotation is used to bind the request attribute to a handler method parameter. Spring retrieves the named attribute's value to populate the parameter annotated with `@RequestAttribute`. While the `@RequestParam` annotation is used bind the parameter values from a query string, `@RequestAttribute` is used to access the objects that have been populated on the server side.

@RequestBody

This annotation is used to annotate request handler method arguments. The `@RequestBody` annotation indicates that a method parameter should be bound to the value of the HTTP request body. The `HttpMessageConverter` is responsible for converting from the HTTP request message to object.

@RequestHeader

This annotation is used to annotate request handler method arguments. The `@RequestHeader` annotation is used to map controller parameter to request header value. When Spring maps the request, `@RequestHeader` checks the header with the name specified within the annotation and binds its value to the handler method parameter. This annotation helps you to get the header details within the controller class.

@RequestParam

This annotation is used to annotate request handler method arguments. Sometimes you get the parameters in the request URL, mostly in GET requests. In that case, along with the `@RequestMapping` annotation, you can use the `@RequestParam` annotation to retrieve the URL parameter and map it to the method argument. The `@RequestParam` annotation is used to bind request parameters to a method parameter in your controller.

@RequestPart

This annotation is used to annotate request handler method arguments. The `@RequestPart` annotation can be used instead of `@RequestParam` to get the content of a specific multipart and bind it to the method argument annotated with `@RequestPart`. This annotation takes into consideration the “Content-Type” header in the multipart (request part).

@ResponseBody

This annotation is used to annotate request handler methods. The `@ResponseBody` annotation is similar to the `@RequestBody` annotation. The `@ResponseBody` annotation indicates that the result type should be written straight in the response body in whatever format you specify like JSON or XML. Spring converts the returned object into a response body by using the `HttpMessageConverter`.

@ResponseStatus

This annotation is used on methods and exception classes. `@ResponseStatus` marks a method or exception class with a status code and a reason that must be returned. When the handler method is invoked the status code is set to the HTTP response which overrides the status information provided by any other means. A controller class can also be annotated with `@ResponseStatus`, which is then inherited by all `@RequestMapping` methods.

@ControllerAdvice

This annotation is applied at the class level. As explained earlier, for each controller, you can use `@ExceptionHandler` on a method that will be called when a given exception occurs. But this handles only those exceptions that occur within the controller in which it is defined. To overcome this problem, you can now use the `@ControllerAdvice` annotation. This annotation is used to define `@ExceptionHandler`, `@InitBinder`, and `@ModelAttribute` methods that apply to all `@RequestMapping` methods. Thus, if you define the `@ExceptionHandler` annotation on a method in a `@ControllerAdvice` class, it will be applied to all the controllers.

@RestController

This annotation is used at the class level. The `@RestController` annotation marks the class as a controller where every method returns a domain object instead of a view. By annotating a class with this annotation, you no longer need to add `@ResponseBody` to all the RequestMapping methods. It means that you no longer use view-resolvers or send HTML in response. You just send the domain object as an HTTP response in the format that is understood by the consumers, like JSON.

`@RestController` is a convenience annotation that combines `@Controller` and `@ResponseBody`.

@RestControllerAdvice

This annotation is applied to Java classes. `@RestControllerAdvice` is a convenience annotation that combines `@ControllerAdvice` and `@ResponseBody`. This annotation is used along with the `@ExceptionHandler` annotation to handle exceptions that occur within the controller.

@SessionAttribute

This annotation is used at method parameter level. The `@SessionAttribute` annotation is used to bind the method parameter to a session attribute. This annotation provides a convenient access to the existing or permanent session attributes.

@SessionAttributes

This annotation is applied at the type level for a specific handler. The `@SessionAttributes` annotation is used when you want to add a JavaBean object into a session. This is used when you want to keep the object in session for short lived. `@SessionAttributes` is used in conjunction with `@ModelAttribute`.

Consider this example:

```
1  @ModelAttribute("person")
2  public Person getPerson() {}
3  // within the same controller as above snippet
4  @Controller
5  @SessionAttributes(value = "person", types = {
6      Person.class
7  })
8  public class PersonController {}
```

The `@ModelAttribute` name is assigned to the `@SessionAttributes` as a value. The `@SessionAttributes` has two elements. The value element is the name of the session in the model and the types element is the type of session attributes in the model.

Spring Cloud Annotations

@EnableConfigServer

This annotation is used at the class level. When developing a project with a number of services, you need to have a centralized and straightforward manner to configure and retrieve the configurations of all the services that you are going to develop. One advantage of using a centralized config server is that you don't need to carry the burden of remembering where each configuration is distributed across multiple and distributed components.

You can use Spring Cloud's `@EnableConfigServer` annotation to start a config server that the other applications can talk to.

@EnableEurekaServer

This annotation is applied to Java classes. One problem that you may encounter while decomposing your application into microservices is that it becomes difficult for every service to know the address of every other service it depends on. There comes the discovery service which is responsible for tracking the locations of all other microservices.

Netflix's Eureka is an implementation of a discovery server and integration is provided by Spring Boot. Spring Boot has made it easy to design a Eureka Server by just annotating the entry class with `@EnableEurekaServer`.

@EnableDiscoveryClient

This annotation is applied to Java classes. In order to tell any application to register itself with Eureka, you just need to add the `@EnableDiscoveryClient` annotation to the application entry point. The application that's now registered with Eureka uses the Spring Cloud Discovery Client abstraction to interrogate the registry for its own host and port.

@EnableCircuitBreaker

This annotation is applied to Java classes that can act as the circuit breaker. The circuit breaker pattern can allow a microservice continue working when a related service fails, preventing the failure from cascading. This also gives the failed service a time to recover.

The class annotated with `@EnableCircuitBreaker` will monitor, open, and close the circuit breaker.

@HystrixCommand

This annotation is used at the method level. Netflix's Hystrix library provides the implementation of a Circuit Breaker pattern. When you apply the circuit breaker to a method, Hystrix watches for the failures of the method. Once failures build up to a threshold, Hystrix opens the circuit so that the subsequent calls also fail. Now Hystrix redirects calls to the method, and they are passed to the specified fallback methods.

Hystrix looks for any method annotated with the `@HystrixCommand` annotation and wraps it into a proxy connected to a circuit breaker so that Hystrix can monitor it.

Consider the following example:

```
1  @Service
2  public class BookService {
3      private final RestTemplate restTemplate;
4      public BookService(RestTemplate rest) {
5          this.restTemplate = rest;
6      }
7      @HystrixCommand(fallbackMethod = "newList") public String bookList()
8      {
9          URI uri = URI.create("http://localhost:8081/recommended");
10         return this.restTemplate.getForObject(uri, String.class);
11     }
12     public String newList() {
13         return "Cloud native Java";
14     }
15 }
```

Here `@HystrixCommand` is applied to the original method `bookList()`. The `@HystrixCommand` annotation has `newList` as the fallback method. So for some reason, if Hystrix opens the circuit on `bookList()`, you will have a placeholder book list ready for the users.

Spring Framework DataAccess Annotations

@Transactional

This annotation is placed before an interface definition, a method on an interface, a class definition, or a public method on a class. The mere presence of `@Transactional` is not enough to activate the transactional behavior. The `@Transactional` is simply metadata that can be consumed by some runtime infrastructure. This infrastructure uses the metadata to configure the appropriate beans with transactional behavior.

The annotation further supports configuration like:

- The Propagation type of the transaction
- The Isolation level of the transaction
- A timeout for the operation wrapped by the transaction
- A read-only flag — a hint for the persistence provider that the transaction must be read only
- The rollback rules for the transaction

Cache-Based Annotations

@Cacheable

This annotation is used on methods. The simplest way of enabling the cache behavior for a method is to annotate it with `@Cacheable` and parameterize it with the name of the cache where the results would be stored.

```
1 @Cacheable("addresses")
2 public String getAddress(Book book) {...}
```

In the snippet above, the method `getAddress` is associated with the cache named `addresses`. Each time the method is called, the cache is checked to see whether the invocation has been already executed and does not have to be repeated.

@CachePut

This annotation is used on methods. Whenever you need to update the cache without interfering the method execution, you can use the `@CachePut` annotation. That is, the method will always be executed and the result cached.

```
1 @CachePut("addresses")
```



```
2 public String getAddress(Book book){...}
```

Using `@CachePut` and `@Cacheable` on the same method is strongly discouraged, as the former forces the execution in order to execute a cache update, the latter causes the method execution to be skipped by using the cache.

@CacheEvict

This annotation is used on methods. It is not that you always want to populate the cache with more and more data. Sometimes, you may want to remove some cache data so that you can populate the cache with some fresh values. In such a case, use the `@CacheEvict` annotation.

```
1 @CacheEvict(value="addresses", allEntries="true")
2 public String getAddress(Book book){...}
```

Here, an additional element, `allEntries`, is used along with the cache name to be emptied. It is set to `true` so that it clears all values and prepares to hold new data.

@CacheConfig

This annotation is a class level annotation. The `@CacheConfig` annotation helps to streamline some of the cache information at one place. Placing this annotation on a class does not turn on any caching operation. This allows you to store the cache configuration at the class level so that you don't have to declare things multiple times.

Task Execution and Scheduling Annotations

@Scheduled

This annotation is a method-level annotation. The `@Scheduled` annotation is used on methods along with the trigger metadata. A method with `@Scheduled` should have a void return type and should not accept any parameters.

There are different ways of using the `@Scheduled` annotation:

```
1 @Scheduled(fixedDelay=5000)
2 public void doSomething() {
3     // something that should execute periodically
4 }
```

In this case, the duration between the end of the last execution and the start of the next execution is fixed. The tasks always wait until the previous one is finished.

```
1 @Scheduled(fixedRate=5000)
2 public void doSomething() {
3     // something that should execute periodically
4 }
```

In this case, the beginning of the task execution does not wait for the completion of the previous execution.

```
1 @Scheduled(initialDelay=1000,fixedRate=5000)
2 public void doSomething() {
3     // something that should execute periodically after an initial delay
4 }
```

The task gets executed initially with a delay and then continues with the specified fixed rate.

@Async

This annotation is used on methods to execute each method in a separate thread. The @Async annotation is provided on a method so that the invocation of that method will occur asynchronously. Unlike methods annotated with @Scheduled, the methods annotated with @Async can take arguments. They will be invoked in the normal way by callers at runtime rather than by a scheduled task.

@Async can be used with both void return type methods and methods that return a value. However, methods with return values must have a Future-typed return value.

Spring Framework Testing Annotations

@BootstrapWith

This annotation is a class-level annotation. The @BootstrapWith annotation is used to configure how the Spring TestContext Framework is bootstrapped. This annotation is used as a metadata to create custom composed annotations and reduce the configuration duplication in a test suite.

@ContextConfiguration

This annotation is a class level annotation that defines a metadata used to determine which configuration files to use to load the ApplicationContext for your test. More specifically @ContextConfiguration declares the annotated classes that will be used to load the context. You can also tell Spring where to locate the file.

```
1 @ContextConfiguration(locations={"example/test-context.xml", loader = Custom  
ContextLoader.class})
```

@WebAppConfiguration

This annotation is a class level annotation. The @WebAppConfiguration is used to declare that the ApplicationContext loaded for an integration test should be a WebApplicationContext. This annotation is used to create the web version of the application context. It is important to note that this annotation must be used with the @ContextConfiguration annotation. The default path to the root of the web application is src/main/webapp. You can override it by passing a different path to the @WebAppConfiguration.

@Timed

This annotation is used on methods. The @Timed annotation indicates that the annotated test method must finish its execution at the specified time period (in milliseconds). If the execution exceeds the specified time in the annotation, the test fails.

```
1 @Timed(millis=10000)  
2 public void testLongRunningProcess() { ... }
```

In this example, the test will fail if it exceeds 10 seconds of execution.

@Repeat

This annotation is used on test methods. If you want to run a test method several times in a row automatically, you can use the @Repeat annotation. The number of times that test method is to be executed is specified in the annotation.

```
1 @Repeat(10)  
2 @Test  
3 public void testProcessRepeatedly() { ... }
```

In this example, the test will be executed 10 times.

@Commit

This annotation can be used as both class-level or method-level annotation. After execution of a test method, the transaction of the transactional test method can be committed using the `@Commit` annotation. This annotation explicitly conveys the intent of the code. When used at the class level, this annotation defines the commit for all test methods within the class. When declared as a method level annotation, `@Commit` specifies the commit for specific test methods overriding the class level commit.

@RollBack

This annotation can be used as both class-level and method-level annotation. The `@RollBack` annotation indicates whether the transaction of a transactional test method must be rolled back after the test completes its execution. If this true, `@Rollback(true)`, the transaction is rolled back. Otherwise, the transaction is committed. `@Commit` is used instead of `@RollBack(false)`.

When used at the class level, this annotation defines the rollback for all test methods within the class.

When declared as a method level annotation, `@RollBack` specifies the rollback for specific test methods overriding the class level rollback semantics.

@DirtiesContext

This annotation is used as both class-level and method-level annotation. `@DirtiesContext` indicates that the Spring ApplicationContext has been modified or corrupted in some manner and it should be closed. This will trigger the context reloading before execution of next test. The ApplicationContext is marked as dirty before or after any such annotated method as well as before or after current test class.

The `@DirtiesContext` annotation supports `BEFORE_METHOD`, `BEFORE_CLASS`, and `BEFORE_EACH_TEST_METHOD` modes for closing the ApplicationContext before a test.

NOTE: Avoid overusing this annotation. It is an expensive operation and if abused, it can really slow down your test suite.

@BeforeTransaction

This annotation is used to annotate void methods in the test class. `@BeforeTransaction` annotated methods indicate that they should be executed before any transaction starts executing. That means the method annotated with `@BeforeTransaction` must be executed before any method annotated with `@Transactional`.

@AfterTransaction

This annotation is used to annotate void methods in the test class. `@AfterTransaction` annotated methods indicate that they should be executed after a transaction ends for test methods. That means the method annotated with `@AfterTransaction` must be executed after the method annotated with `@Transactional`.

@Sql

This annotation can be declared on a test class or test method to run SQL scripts against a database. The `@Sql` annotation configures the resource path to SQL scripts that should be executed against a given database either before or after an integration test method. When `@Sql` is used at the method level it, will override any `@Sql` defined in at class level.

@SqlConfig

This annotation is used along with the `@Sql` annotation. The `@SqlConfig` annotation defines the metadata that is used to determine how to parse and execute SQL scripts configured via the `@Sql` annotation. When used at the class level, this annotation serves as global configuration for all SQL scripts within the test class. But when used directly with the `config` attribute of `@Sql`, `@SqlConfig` serves as a local configuration for SQL scripts declared.

@SqlGroup

This annotation is used on methods. The `@SqlGroup` annotation is a container annotation that can hold several `@Sql` annotations. This annotation can declare nested `@Sql` annotations. In addition, `@SqlGroup` is used as a meta-annotation to create custom composed annotations. This annotation can also be used along with repeatable annotations, where `@Sql` can be declared several times on the same method or class.

@SpringBootTest

This annotation is used to start the Spring context for integration tests. This will bring up the full autoconfiguration context.

@DataJpaTest

The `@DataJpaTest` annotation will only provide the autoconfiguration required to test Spring Data JPA using an in-memory database such as H2.

This annotation is used instead of `@SpringBootTest`

@DataMongoTest

The `@DataMongoTest` will provide a minimal autoconfiguration and an embedded MongoDB for

running integration tests with Spring Data MongoDB.

@WebMVCTest

The @WebMVCTest will bring up a mock servlet context for testing the MVC layer. Services and components are not loaded into the context. To provide these dependencies for testing, the @MockBean annotation is typically used.

@AutoConfigureMockMVC

The @AutoConfigureMockMVC annotation works very similarly to the @WebMVCTest annotation, but the full Spring Boot context is started.

@MockBean

Creates and injects a Mockito Mock for the given dependency.

@JsonTest

Will limit the auto-configuration of Spring Boot to components relevant to processing JSON.

This annotation will also autoconfigure an instance of JacksonTester or GsonTester.

@TestPropertySource

Class level annotation used to specify property sources for the test class.

Build vs Buy a Data Quality Solution: Which is Best for You? Maintaining high quality data is essential for operational efficiency, meaningful analytics and good long-term customer relationships. But, when dealing with multiple sources of data, data quality becomes complex, so you need to know when you should build a custom data quality tools effort over canned solutions. Download our whitepaper for more insights into a hybrid approach.

Like This Article? Read More From DZone



**AutoValue: Generated
Immutable Value Classes**




An Introduction to Spring



Spring Component Scan



**Free DZone Refcard
Core Java Concurrency**

Published at DZone with permission of John Thompson, DZone MVB. [See the original article here.](#) 

Opinions expressed by DZone contributors are their own.

Java Partner Resources

Get Started with Spring Security 5.0 and OpenID Connect (OIDC)

Okta



Get Unit Testing Done Right: Top Tips for Java Developers

Parasoft



Top 5 Java Performance Metrics, Tips and Tricks

AppDynamics



Level up your code with a Pro IDE

JetBrains

