# Testing ScanPath

jim stafford

Fall 2025 v2025-06-29: Built: 2026-02-14 08:33 EST

# Table of Contents

# Chapter 1. Introduction

Spring Boot is a large collection of conventions. These conventions can be changed, but changes can have slight ramifications for tests. This lecture will focus on the conventions surrounding component scan paths and the impacts they have for testing. It is good to know why and how conventions are enabled, when these conventions are disrupted, and how they can be retained.

## 1.1. Goals

The student will learn:

- the Spring Boot Java package conventions
- the Spring Boot test conventions
- mechanisms used to customize Spring Boot conventions for use with non-compliant project aspects

## 1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. identify the relevant annotations used to enact Spring Boot component scanpath conventions
2. identify Spring Boot Java package conventions
3. identify the purpose of `@TestComponent` in excluding automatic addition to application context
4. identify the proper way to override the default component scanpath and the potential ramifications of some options

# Chapter 2. Relevant Spring Boot Annotations

Before going too much further, I want to point out that:

- when I say that something *"looks for @SpringBootApplication"*, it is really looking for the singleton `@SpringBootConfiguration` that `@SpringBootApplication` includes
- when I say *"components in the scan path"*, by default, this includes all classes with annotations that include `@Component`. That can mean both the `src/main` and `src/test` trees during testing.
- the `@SpringBootApplication` (technically the `@SpringBootConfiguration`) is searched for to identify the default component scan path that can locate components.

## 2.1. @SpringBootApplication

`@SpringBootApplication` is:

- a `@SpringBootConfiguration`. A single class with the `@SpringBootConfiguration` annotation is required to be in every Spring Boot application.
- enables processing `@Configuration` classes identified in the `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` file of each dependency.
- identifies the base package for the component scan (default is current Java package)
- identifies include and exclude filters to process classes found during component scan

The following snippet shows a few annotations included and configured by `@SpringBootApplication`.

*@SpringBootApplication includes @SpringConfiguration*

```
...
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class)
})
public @interface SpringBootApplication {
```

## 2.2. @ComponentScan

`@ComponentScan` defines high-level and some low-level details about how components are found and conditionally added to the application context. The three properties within scope of this lecture are:

- **basePackages** - identifies the root Java packages to start scans from. This can be String-based or Java class-based. If providing a Java class, only the Java package for the class is relevant. The specific class has no special meaning (i.e., it defines a root where its siblings and children are all scanned)

- **includeFilters** - specifies additional filters that are added to the base set (that identify `@Component` packages) of filters
- **excludeFilters** - specifies filters that eliminate candidate classes

A point of emphasis in this lecture will be that the default `@ComponentScan` defined by `@SpringBootApplication` will enact standard conventions. If certain properties are adjusted or a `@ComponentScan` is expressed, this can impact other code that are trying to follow conventions. Factor in these conventions if modifying the default `@ComponentScan`.

## 2.3. @SpringBootConfiguration

`@SpringBootConfiguration` includes the `@Configuration` annotation, which makes classes using it capable of defining application context details, like `@Bean` factories, configuration sources, and component scan paths. This special annotation allows for there to be a single instance detected in the application context.

*@SpringBootConfiguration includes @Configuration*

```
...
@Configuration
public @interface SpringBootConfiguration {
```

## 2.4. @AutoConfiguration

`@AutoConfiguration` includes the `@Configuration` annotation, which of course allows it to be a `@Configuration`/`@Component`. However, this specialization allows the class to be excluded from other `@Component` classes found during normal component scan.

Classes with the `@AutoConfiguration` annotations are meant to only be added when referenced by an `AutoConfiguration.imports` file entry in a dependency. The default `AutoConfigurationExcludeFilter` put in place by `@SpringBootApplication` implements that convention.

*@AutoConfiguration Processed by AutoConfigurationExcludeFilter*

```
@ComponentScan(excludeFilters = {
  @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
  @Filter(type = FilterType.CUSTOM,
          classes = AutoConfigurationExcludeFilter.class)//used to special process
AutoCfg
})
public @interface SpringBootApplication {
```

`@AutoConfiguration` is unique to auto-configuration and, by convention, should only be processed under those conditions.

*@AutoConfiguration includes @Configuration*

```
@Configuration(proxyBeanMethods = false)
```

```
@AutoConfigureBefore
@AutoConfigureAfter
public @interface AutoConfiguration {
```

## 2.5. @Configuration

`@Configuration` includes the `@Component` annotation, which allows it to be found with all other `@Component` classes during the component scan. Again, this can be in the `src/main` or `src/test` (during testing) tree and any other dependency in the component scan path

*@Configuration includes @Component*

```
...
@Component
public @interface Configuration {
```

## 2.6. @TestConfiguration

A specialized `@Configuration` that enables customizations unique for testing.

```
@Configuration
@TestComponent
public @interface TestConfiguration {
```

Classes with this customization are meant to change things for a test. They can be:

- explicitly `@Imported` by name

```
//ImportedExternalConfigNTest.java
@SpringBootTest
@Import(ExternalTestConfiguration.class)
public class ImportedExternalConfigNTest {

//ExternalTestConfiguration.java
@TestConfiguration
public class ExternalTestConfiguration {
```

- implicitly imported if defined in a static internal class of the `@SpringBootTest`

```
@SpringBootTest
public class EmbeddedTestConfigNTest {
    @TestConfiguration
    static class EmbeddedTestConfiguration {
```

When following conventions, `@TestComponents` are not automatically picked up as a `@Component` by

the component scan.

## 2.7. @TestComponent

`@TestCompnent` identifies `@Component` classes that should only be used during test. This convention is implemented through the `TypeExcludeFilter` put in place by `@SpringBootApplication` in the `@ComponentScan`.

*@SpringBootApplication default @ComponentScan Definition*

```
@ComponentScan(excludeFilters = {
  @Filter(type = FilterType.CUSTOM,
          classes = TypeExcludeFilter.class), //used to exclude @TestComponents
  @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class)
})
public @interface SpringBootApplication {
```

The `TypeExcludeFilter` plays 2 roles.

- an aggregate to place all identified `TypeExcludeFilter` classes into
- a base class to implement a filter

If the `TypeExcludeFilter` is missing from the `@ComponentScan`, then adhoc filters from test frameworks like `@SpringBootTest` or test slices like `@JdbcTest` will not work as intended.

## 2.8. @Component

`@Component` is an annotation that, when applied to a class, makes that class elible to be picked up by a component scan.

The following shows a relationship summary of the annotations and conventions we have just discussed.

```
@Component
  `-- @Configuration
  |   `-- @SpringBootConfiguration
  |   |   `-- @SpringBootApplication
  |   `-- @AutoConfiguration -- triggers selective exclusion based in imports
  `-- @TestComponent -- triggers exclusion
      `-- @TestConfiguration
```

# Chapter 3. Java Package Conventions

A conventional Java package layout in Spring Boot has:

- class defining `@SpringBootApplication` at or above the Java package(s) that define the components within that application
- test classes using `@SpringBootTest` to be at or below the Java package that defines the `@SpringBootConfiguration` — the parent of `@SpringBootConfiguration`

## 3.1. Conventional Spring Boot Java Package Layout

The following figure shows an example of a Java package layout that complies with those standard Spring Boot conventions.

*Conventional Spring Boot Java Package Layout*

```
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- info
    |           `-- ejava
...
    |               `-- conventional
    |                   |-- Application.java //@SpringBootApplication
    |                   `-- cmp
    |                       `-- MyComponent.java //@Component
    `-- test
        `-- java
            `-- info
                `-- ejava
...
                    `-- conventional
                        |-- AtNoTestConfigNTest.java //@SpringBootTest
                        `-- child
                            `-- ChildNoTestConfigNTest.java //@SpringBootTest
```

## 3.2. Conventional @SpringBootApplication

A "main" class complying with the Java package conventions can simply define `@SpringBootApplication`, with no extra properties.

*Conventional @SpringBootApplication*

```
package info.ejava.examples.app.testing.scanpath.conventional;
...
//component(s) are at or below this class' package
@SpringBootApplication
```

```
public class Application {
    public static void main(String...args) {
        SpringApplication.run(Application.class, args);
    }
}
```

This:

- sets the component scan basePackages to this class' Java package
- sets the component scan filters to their default and conventional settings

*Default, Conventional @SpringBootApplication Settings for @ComponentScan*

```
@ComponentScan(excludeFilters = {
  @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
  @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class)
})
```

## 3.3. Conventional @Component Location

The components found at or below the Java package of the class defining the `@SpringBootApplication` are automatically included in the application context. This can be `@Component` or `@Bean` factories within a `@Configuration` class.

*@Component Class Found below Java Package of @SpringBootApplication*

```
package info.ejava.examples.app.testing.scanpath.conventional.cmp;
... //Java package is at or below class with @SpringBootApplication
@Component
@Data
@AllArgsConstructor
public class MyComponent {
    private String source="from src/main";
}
```

## 3.4. Conventional @SpringBootTest

Test classes using `@SpringBootTest` that are at or below the `@SpringBootConfiguration` class are able to locate the single instance using default Java package searching.

```
package info.ejava.examples.app.testing.scanpath.conventional.child;
...
@SpringBootTest //SpringBootApplication is found at or above this Java package
public class ChildNoTestConfigNTest {
    @Autowired
    private MyComponent theComponent;
```

```
    @Test
    void from_src_main() {
        then(theComponent.getSource()).isEqualTo("from src/main");
    }
}
```

# Chapter 4. @TestConfiguration

`@SpringBootTest` will run with the `@ComponentScan` as defined by its referenced `@SpringBootConfiguration`/`@SpringBootApplication`. The default `@ComponentScan` will have a filter in place that will allow `@SpringBootTest` to keep `@TestComponent` from being found. A `@TestComponent` is only included when it complies with certain rules or explicitly imported by the test.

## 4.1. Imported External @TestConfiguration

You can create a `@TestConfiguration` that can be shared among tests, but each one must explicitly include the class using an `@Import`.

*Sharable @TestConfiguration Class*

```
@TestConfiguration
public class ExternalTestConfiguration {
    @Bean
    MyComponent myComponent() {
        return new MyComponent("from ExternalTestConfiguration");
    }
}
```

*@SpringBootTest using External @TestConfiguration*

```
@SpringBootTest
@Import(ExternalTestConfiguration.class)
public class ImportedExternalConfigNTest {
    @Autowired
    private MyComponent theComponent;

    @Test
    void from_external_test_configuration() {
        then(theComponent.getSource()).isEqualTo("from ExternalTestConfiguration");
    }
}
```

## 4.2. Embedded @TestConfiguration

Configuration settings unique to the class can be expressed within the test class using a static, embedded `@TestConfiguration` class.

*Static, Embedded @TestConfiguration found by Default App Reference*

```
//@SpringBootTest
public class EmbeddedTestConfigNTest {
    @Autowired
    private MyComponent theComponent;
```

```
    @TestConfiguration
    static class EmbeddedTestConfiguration {
        @Bean
        MyComponent myComponent() {
            return new MyComponent("from EmbeddedTestConfiguration");
        }
    }

    @Test
    void from_embedded_test_configuration() {
        then(theComponent.getSource()).isEqualTo("from EmbeddedTestConfiguration");
    }
}
```

## 4.3. Embedded @TestConfiguration References

If the `@SpringBootTest` uses

- standard "at or above" search logic for the `@SpringBootConfiguration` class, any static class annotated with `@TestConfiguration` and embedded within the test class will be automatically located.

  *Static, Embedded @TestConfiguration found by Default App Reference*

  ```
  @SpringBootTest
  public class EmbeddedTestConfigNTest {
  ```

- an explicit reference to the `@SpringBootConfiguration`

  *Explicit @Import of @TestConfiguration*

  ```
  @SpringBootTest(classes = Application.class)
  //embedded scanning disabled when @SpringBootTest.classes set
  //we must explicitly import the nested test configuration
  @Import(EmbeddedTestConfigNTest.EmbeddedTestConfiguration.class)
  public class EmbeddedTestConfigNTest {
  ```

## 4.4. @Importing Embedded @TestConfiguration

Lacking the explicit `@Import` when using `@SpringBootTest.classes`, causes the embedded `@TestConfiguration` to be ignored and the `@Component` from `src/main` gets used.

*Assertion Error when Missing @Import*

```
org.opentest4j.AssertionFailedError:
expected: "from EmbeddedTestConfiguration"
 but was: "from src/main"
```

Using the embedded static class `@Import` along with `@SpringBootTest.classes` is allowed and can provide a path to working in most cases until the component scan is changed.

Up until this point, we have used the default, conventional `@ComponentScan`.

# Chapter 5. @SpringBootApplication Not Root

Sometimes conventions get broken and the `@SpringBootApplication` class does not get positioned at the root of all components placed into the component scanpath.

*@SpringBootApplication not at root of @Components*

```
    `-- scanpath
        `-- scanbasepackages
            |-- app
            |   `-- Application.java //@SpringBootApplication
            `-- cmp
                `-- MyComponent.java //@Component
```

## 5.1. @SpringBootApplication.scanBasePackages

We can solve this by setting the `SpringBootApplication.scanBasePackages` property. As you will see later, this approach inflicts the least amount of pain. This approach only changes the base packages scanned. It does not change the `@ComponentScan` filters.

```
package info.ejava.examples.app.testing.scanpath.scanbasepackages.app;
...
@SpringBootApplication(
    scanBasePackages = "info.ejava.examples.app.testing.scanpath.scanbasepackages"
)
public class Application {
    public static void main(String...args) {
        SpringApplication.run(Application.class, args);
    }
}
```

## 5.2. @ComponentScan.basePackages — WRONG WAY

Remember that the `@SpringBootApplication.scanBasePackages` is an alias for `@ComponentScan.basePackages`. That means some may attempt to solve this my supplying a `@ComponentScan` with only the `basePackages` property.

```
@ComponentScan(basePackages =
"info.ejava.examples.app.testing.scanpath.scanbasepackages")
```

The problem with that approach is that it — wipes out the default `excludeFilters` put in place by `@SpringBootApplication`. The elimination of that set of exclude filters causes `@TestComponents` in the classpath to be found — including embedded ones in separate test classes.

```
org.springframework.beans.factory.support.BeanDefinitionOverrideException: Invalid
```

```
bean definition with name 'myComponent' defined in class path resource
[info/ejava/examples/app/testing/scanpath/componentscan/app/EmbeddedTestConfigNTest$Em
beddedTestConfiguration.class]

since there is already; defined in class path resource
[info/ejava/examples/app/testing/scanpath/componentscan/altpath/AltPackageEmbeddedTest
ConfigNTest$EmbeddedTestConfiguration.class]] bound.
```

## 5.3. @ComponentScan — Right Way

Spring Boot documentation [1] insists that if you are going to override `@ComponentScan`, make sure to re-enable the `excludeFilters`

- **TypeExcludeFilter** - acts as an aggregate filter allowing things like test frameworks to exclude classes annotated with `@TestComponent`
- **AutoConfigurationExcludeFilter** - verifies classes annotated with `@AutoConfiguration` are only included if they are referenced by a `AutoConfiguration.imports` file.

*Correct Use of @ComponentScan*

```
@SpringBootApplication
@ComponentScan(basePackages = "info.ejava.examples.app.testing.scanpath.componentscan
",
  excludeFilters = {
    @ComponentScan.Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class
),
    @ComponentScan.Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class)
    }
)
public class Application {
    public static void main(String...args) {
        SpringApplication.run(Application.class, args);
    }
}
```

[1] *"Detecting Test Configuration"*,"docs.spring.io"

# Chapter 6. Custom TypeExcludeFilter

We can create a custom implementation of `TypeExcludeFilter` that can remove any unwanted `@Component` in our test.

## 6.1. MyTypeExcludeFilter

The following snippet shows a custom `TypeExcludeFilter` that will exclude the `MyComponent` from the `src/main` tree.

```
package info.ejava.examples.app.testing.scanpath.componentscan.app;

import info.ejava.examples.app.testing.scanpath.componentscan.cmp.MyComponent;
...
@EqualsAndHashCode(callSuper = false)
public class MyTypeExcludeFilter extends TypeExcludeFilter {
    @Override
    public boolean match(MetadataReader metadataReader,
                MetadataReaderFactory metadataReaderFactory) throws IOException {
        return metadataReader.getClassMetadata().getClassName()
            .equals(MyComponent.class.getName());
    }
}
```

## 6.2. @TypeExcludeFilters

We can register the custom filter for our specific test(s) by adding a `@TypeExcludeFilters` annotation naming our custom class.

The following snippet shows a custom `TypeExcludeFilter` being registered and expecting that no component satisfy the injection. Since that is the only component of type `MyComponent` and the filter is designed to exclude it — nothing gets injected. A better example might be an amiguous match and excluding the one not wanted.

```
@SpringBootTest //Application class is in package at or above
@TypeExcludeFilters(MyTypeExcludeFilter.class)
public class MyTypeExcludeFilterNTest {
    @Autowired(required = false)
    private MyComponent theComponent;

    @Test
    void no_component() {
        then(theComponent).isNull();
    }
}
```

# Chapter 7. Summary

In this module we learned:

1. relevant annotations used to enact Spring Boot component scanpath conventions
2. Spring Boot Java package conventions
3. the purpose of `@TestComponent` in excluding automatic addition to application context
4. the proper way to override the default component scanpath and the potential ramifications of some options