# 1. @SpringBootApplication

***@SpringBootApplication*** automatically provides the features of the following annotations:

*@Configuration @EnableAutoConfiguration @ComponentScan*

*@EnableAutoConfiguration* annotation tells Spring Boot to "guess" how you will want to configure Spring, based on the jar dependencies that you have added. For example, If HSQLDB is on your classpath, and you have not manually configured any database connection beans, then Spring will auto-configure an in-memory database.

One of the main advantages of Spring Boot is its annotation driven versus traditional xml based configurations, @EnableAutoConfiguration automatically configures the Spring application based on its included jar files, it sets up defaults or helper based on dependencies in pom.xml. Auto-configuration is usually applied based on the classpath and the defined beans. Therefore, we donot need to define any of the DataSource, EntityManagerFactory, TransactionManager etc and magically based on the classpath, Spring Boot automatically creates proper beans and registers them for us. For example when there is a tomcat-embedded.jar on your classpath you likely need a TomcatEmbeddedServletContainerFactory (unless you have defined your own EmbeddedServletContainerFactory bean). @EnableAutoConfiguration has a exclude attribute to disable an auto-configuration explicitly otherwise we can simply exclude it from the pom.xml, for example if we do not want Spring to configure the tomcat then exclude spring-bootstarter-tomcat from spring-boot-starter-web, or DataSourceAutoConfiguration.class to exlude db configuration.

*@ComponentScan* tells Spring to look for other components, configurations, and services in the specified package. Spring is able to auto scan, detect and register your beans or components from pre-defined project package. If no package is specified current class package is taken as the root package.

If you want to get the maximum advantage of spring boot’s auto configuration feature, it is expected to put all your class packages under spring boot main application package (directly in main package or indirectly as sub packages).

If you dont use *@SpringBootApplication*, but you try defining your own annotation then spring boot will not take care of internal auto configurations so this is the reason we have to enable repositories.

In order to let spring knows what DataSource is related to what Repository you should define it at the @EnableJpaRepositories annotation.. Let's assume that we have two entities, the Servers entity and the Domains entity and each one has its own Repo then each Repository has its own JpaDataSource configuration.

1. Group all the repositories based on the Data Source that they are related to. For example

Repository for Domains entities (package: *org.springdemo.multiple.datasources.repository.domains):*

Repository for Servers entities (package: *org.springdemo.multiple.datasources.repository.servers)*

2.Define a configuration and set one Datasource as primary

For each JPA Data Soruce you need to define a configuration, in this example I show how to configure two different DataSources. In order to avoid the error message: Parameter 0 of constructor in org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration required a single bean, but 2 were found: just set one of the datasource as @Primary, in this example I select the Servers Datasource as primary:

@Configuration

@EnableJpaRepositories(

entityManagerFactoryRef = "serversEntityManager",

transactionManagerRef = "serversTransactionManager",

basePackages = {"org.springdemo.multiple.datasources.repository.servers"}

)

public class ServersConfig {

@Bean(name = "serversEntityManager")

public LocalContainerEntityManagerFactoryBean getServersEntityManager(EntityManagerFactoryBuilder builder,

@Qualifier("serversDataSource") DataSource serversDataSource){

return builder

.dataSource(serversDataSource)

.packages("org.springdemo.multiple.datasources.domain.servers")

.persistenceUnit("servers")

.properties(additionalJpaProperties())

.build();

}

Map<String,?> additionalJpaProperties(){

Map<String,String> map = new HashMap<>();

map.put("hibernate.hbm2ddl.auto", "create");

map.put("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");

map.put("hibernate.show\_sql", "true");

return map;

}

@Bean("serversDataSourceProperties")

@Primary

@ConfigurationProperties("app.datasource.servers")

public DataSourceProperties serversDataSourceProperties(){

return new DataSourceProperties();

}

@Bean("serversDataSource")

@Primary

@ConfigurationProperties("app.datasource.servers")

public DataSource serversDataSource(@Qualifier("serversDataSourceProperties") DataSourceProperties serversDataSourceProperties) {

return serversDataSourceProperties().initializeDataSourceBuilder().build();

}

@Bean(name = "serversTransactionManager")

public JpaTransactionManager transactionManager(@Qualifier("serversEntityManager") EntityManagerFactory serversEntityManager){

JpaTransactionManager transactionManager = new JpaTransactionManager();

transactionManager.setEntityManagerFactory(serversEntityManager);

return transactionManager;

}

}

@Configuration

@EnableJpaRepositories(

entityManagerFactoryRef = "domainsEntityManager",

transactionManagerRef = "domainsTransactionManager",

basePackages = {"org.springdemo.multiple.datasources.repository.domains"}

)

public class DomainsConfig {

@Bean(name = "domainsEntityManager")

public LocalContainerEntityManagerFactoryBean getdomainsEntityManager(EntityManagerFactoryBuilder builder

,@Qualifier("domainsDataSource") DataSource domainsDataSource){

return builder

.dataSource(domainsDataSource)

.packages("org.springdemo.multiple.datasources.domain.domains")

.persistenceUnit("domains")

.properties(additionalJpaProperties())

.build();

}

Map<String,?> additionalJpaProperties(){

Map<String,String> map = new HashMap<>();

map.put("hibernate.hbm2ddl.auto", "create");

map.put("hibernate.dialect", "org.hibernate.dialect.H2Dialect");

map.put("hibernate.show\_sql", "true");

return map;

}

@Bean("domainsDataSourceProperties")

@ConfigurationProperties("app.datasource.domains")

public DataSourceProperties domainsDataSourceProperties(){

return new DataSourceProperties();

}

@Bean("domainsDataSource")

public DataSource domainsDataSource(@Qualifier("domainsDataSourceProperties") DataSourceProperties domainsDataSourceProperties) {

return domainsDataSourceProperties.initializeDataSourceBuilder().build();

}

@Bean(name = "domainsTransactionManager")

public JpaTransactionManager transactionManager(@Qualifier("domainsEntityManager") EntityManagerFactory domainsEntityManager){

JpaTransactionManager transactionManager = new JpaTransactionManager();

transactionManager.setEntityManagerFactory(domainsEntityManager);

return transactionManager;

}

}

In order to separate each datasource I put the configuration in the application.properties file, like this

app.datasource.domains.url=jdbc:h2:mem:~/test

app.datasource.domains.driver-class-name=org.h2.Driver

app.datasource.servers.driver-class-name=com.mysql.jdbc.Driver

app.datasource.servers.url=jdbc:mysql://localhost:3306/v?autoReconnect=true&useSSL=false

app.datasource.servers.username=myuser

app.datasource.servers.password=mypass

# 2. @SpringBootTest

Spring Boot provides the @SpringBootTest annotation which we can use to create an application context containing all the objects we need for all of the above test types. Note, however, that overusing @SpringBootTest might lead to very long-running test suites,

For tests that cover integration with the web layer or persistence layer, we can use @WebMvcTest or @DataJpaTest instead. Tests that cover the whole Spring Boot application from incoming request to database, or tests that cover certain parts of the application that are hard to set up manually, we can and should use @SpringBootTest.

@SpringBootTest by default starts searching in the current package of the test class and then searches upwards through the package structure, looking for a class annotated with @SpringBootConfiguration from which it then reads the configuration to create an application context. This class is usually our main application class since the @SpringBootApplication annotation includes the @SpringBootConfiguration annotation. It then creates an application context very similar to the one that would be started in a production environment.

**@ExtendWith(SpringExtension.**class**)**

**@SpringBootTest**

**@AutoConfigureMockMvc**

**class** **RegisterUseCaseIntegrationTest** **{**

**@Autowired**

**private** **MockMvc** mockMvc**;**

**@Autowired**

**private** **ObjectMapper** objectMapper**;**

**@Autowired**

**private** **UserRepository** userRepository**;**

**@Test**

**void** **registrationWorksThroughAllLayers()** **throws** **Exception** **{**

**UserResource** user **=** **new** **UserResource(**"Zaphod"**,** "zaphod@galaxy.net"**);**

mockMvc**.**perform**(**post**(**"/forums/{forumId}/register"**,** 42L**)**

**.**contentType**(**"application/json"**)**

**.**param**(**"sendWelcomeMail"**,** "true"**)**

**.**content**(**objectMapper**.**writeValueAsString**(**user**)))**

**.**andExpect**(**status**().**isOk**());**

**UserEntity** userEntity **=** userRepository**.**findByName**(**"Zaphod"**);**

assertThat**(**userEntity**.**getEmail**()).**isEqualTo**(**"zaphod@galaxy.net"**);**

**}**

**}**

The code examples in this tutorial use the @ExtendWith annotation to tell JUnit 5 to enable Spring support. As of Spring Boot 2.1, we no longer need to load the SpringExtension because it's included as a meta annotation in the Spring Boot test annotations like @DataJpaTest, @WebMvcTest, and @SpringBootTest.

Here, we additionally use @AutoConfigureMockMvc to add a MockMvc instance to the application context.We use this MockMvc object to perform a POST request to our application and to verify that it responds as expected. @WebMvcTest auto-configures MockMvc. Mock MVC offers a powerful way to quickly test MVC controllers without needing to start a full HTTP server. If you need to register extra components, such as the Jackson Module, you can import additional configuration classes by using @Import on your test. You can also auto-configure MockMvc in a non-@WebMvcTest (such as @SpringBootTest) by annotating it with @AutoConfigureMockMvc.

If many of our tests need the same set of properties, we can create a configuration file application-<profile>.properties or application-<profile>.yml and load the properties from that file by activating a certain profile:

**@SpringBootTest**

**@ActiveProfiles(**"test"**)**

**class** **SpringBootProfileTest** **{**

**@Value(**"${foo}"**)**

**String** foo**;**

**@Test**

**void** **test(){**

assertThat**(**foo**).**isEqualTo**(**"bar"**);**

**}**

**}**

Another way to customize a whole set of properties is with the @TestPropertySource annotation. All properties from the foo.properties file are loaded into the application context

**@SpringBootTest**

**@TestPropertySource(**locations **=** "/foo.properties"**)**

**class** **SpringBootPropertySourceTest** **{**

**@Value(**"${foo}"**)**

**String** foo**;**

**@Test**

**void** **test(){**

assertThat**(**foo**).**isEqualTo**(**"bar"**);**

**}**

**}**

If certain beans are not included in the default application context, but we need them in a test, we can import them using the @Import annotation. By default, a Spring Boot application includes all components it finds within its package and sub-packages, so this will usually only be needed if we want to include beans from other packages.

**package** other**.**namespace**;**

**@Component**

**public** **class** **Foo** **{**

**}**

**@SpringBootTest**

**@Import(**other**.**namespace**.**Foo**.**class**)**

**class** **SpringBootImportTest** **{**

**@Autowired**

**Foo** foo**;**

**@Test**

**void** **test()** **{**

assertThat**(**foo**).**isNotNull**();**

**}**

**}**

We can even create a whole custom Spring Boot application to start up in tests. If this application class is in the same package as the real application class, but in the test sources rather than the production sources, @SpringBootTest will find it before the actual application class and load the application context from this application instead. Alternatively, we can tell Spring Boot which application class to use to create an application context:

**@SpringBootTest(**classes **=** **CustomApplication.**class**)**

**class** **CustomApplicationTest** **{**

**}**

Sometimes you need to add extra Spring configuration (additional beans for instance), or alter the existing configuration with mocked bean instances. You can use an annotated @TestConfiguration class for that. You’ll have to point out the additional configuration class in the @SpringBootTest annotation. Be careful to not use @Configuration instead of @TestConfiguration. @Configuration breaks the spring boot autoconfiguration mecanism.

When doing this, however, we’re testing an application context that may be completely different from the production environment, so this should be a last resort only when the production application cannot be started in a test environment. Usually, there are better ways, though, such as to make the real application context configurable to exclude beans that won’t start in a test environment. Let’s look at this in an example. Let’s say we use the @EnableScheduling annotation on our application class. Each time the application context is started (even in tests), all @Scheduled jobs will be started and may conflict with our tests. We usually don’t want the jobs to run in tests, so we can create a second application class without the @EnabledScheduling annotation and use this in the tests. However, the better solution would be to create a configuration class that can be toggled with a property:

**@Configuration**

**@EnableScheduling**

**@ConditionalOnProperty(**

name **=** "io.reflectoring.scheduling.enabled"**,**

havingValue **=** "true"**,**

matchIfMissing **=** **true)**

**public** **class** **SchedulingConfiguration** **{**

**}**

We have moved the @EnableScheduling annotation from our application class to this special confgiuration class. Setting the property io.reflectoring.scheduling.enabled to false will cause this class not to be loaded as part of the application context:

**@SpringBootTest(**properties **=** "io.reflectoring.scheduling.enabled=false"**)**

**class** **SchedulingTest** **{**

**@Autowired(**required **=** **false)**

**private** **SchedulingConfiguration** schedulingConfiguration**;**

**@Test**

**void** **test()** **{**

assertThat**(**schedulingConfiguration**).**isNull**();**

**}**

**}**

# 3. @ContextConfiguration

@ContextConfiguration loads an ApplicationContext for Spring integration test. @ContextConfiguration can load ApplicationContext using XML resource or the JavaConfig annotated with @Configuration. The @ContextConfiguration annotation can also load a component annotated with @Component, @Service, @Repository etc. We can also load classes annotated with javax.inject.

@ContextConfiguration annotation has following elements.

* classes: The classes annotated with @Configuration are assigned to load ApplicationContext.
* inheritInitializers: A Boolean value to decide whether context initializers from test super classes should be inherited or not. Default is true.
* inheritLocations: A Boolean value to decide whether resource locations or annotated classes from test super classes should be inherited or not. Default value is true.
* initializers: We specify application context initializer classes that initialize ConfigurableApplicationContext.
* loader: We specify our ContextLoader or SmartContextLoader class to load ApplicationContext.
* locations: We specify resource locations to load ApplicationContext.
* name: Name of context hierarchy level represented by this configuration.
* value: It is the alias for locations element.

@ContextConfiguration(classes = AppConfig.class)

public class MyAppTest {

------

}

@ContextConfiguration(locations= "/spring-config.xml")

public class MyAppTest2 {

------

}

If you try to use both locations and classes attributes then spring will throw an error, only one of them needs to be specified. If you don’t specify the classes attribute, the test class will automatically load any static inner components defined using spring annotations.

@SpringBootTest is relevant only if you have a Spring Boot application. This framework uses Spring under the hood but, in a nutshell, comes with many pre-defined recipes/practices of how to write the "infrastructure" of the application: - configuration management, - package structure, - pluggability - logging - database integration etc. So Spring Boot establishes well-defined processes to deal with all the aforementioned items, and if you want to start the test that will mimic the spring boot application, then you use @SpringBootTest annotation. Otherwise (or in case you have only spring driven application and not a spring boot) - don't use it at all. @ContextConfiguration just says what beans would you like to use in Spring driven application (it also works with spring boot).