1 Sources

- https://redips789.github.io/spring-certification/Spring-Certification.html
- https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring
- https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring
- https://www.baeldung.com/spring-bean-names
- https://www.baeldung.com/spring-core-annotations
- https://www.baeldung.com/spring-bean-annotations
- https://www.baeldung.com/spring-component-scanning
- https://www.baeldung.com/spring-annotations-resource-inject-autowire
- https://www.digitalocean.com/community/tutorials/spring-bean-life-cycle

2 Bean Lifecycle

2.1 Overview

From a bird's eye, everything that happens before a bean is ready to use can be assigned to one of three phases (see fig. 1):

- Loading and maybe modifying bean definitions
- Instantiating beans
- Initializing beans

Figure 2 focuses on pre-initialization.
On the other hand, fig. 4 zooms in on post-instantiation.
See https://www.digitalocean.com/community/tutorials/spring-bean-life-cycle for code to display the order of invocations.

2.1.1 Load bean definitions, creating an ordered graph

In this step, all the configuration files – @Configuration classes or XML files – are processed. For annotation-based configuration, all the classes annotated with @Components are scanned to load the bean definitions.

Bean definitions are passed to a BeanFactory, each under its id and type. For example, ApplicationContext is a BeanFactory.

Then, BeanFactoryPostProcessors are run.

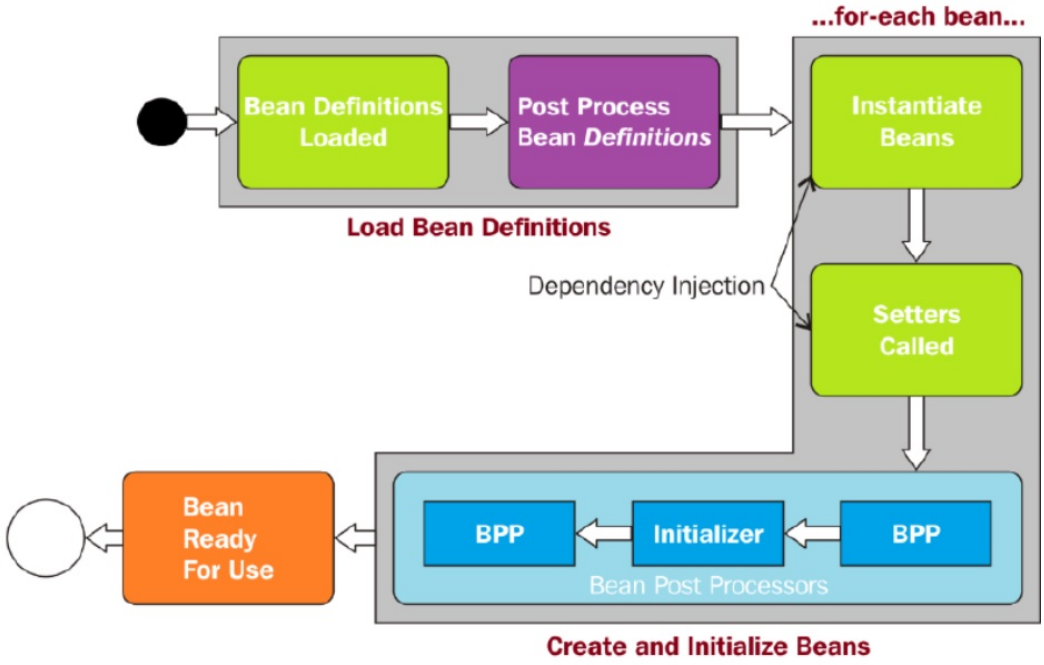


Figure 1: Lifecycle overview

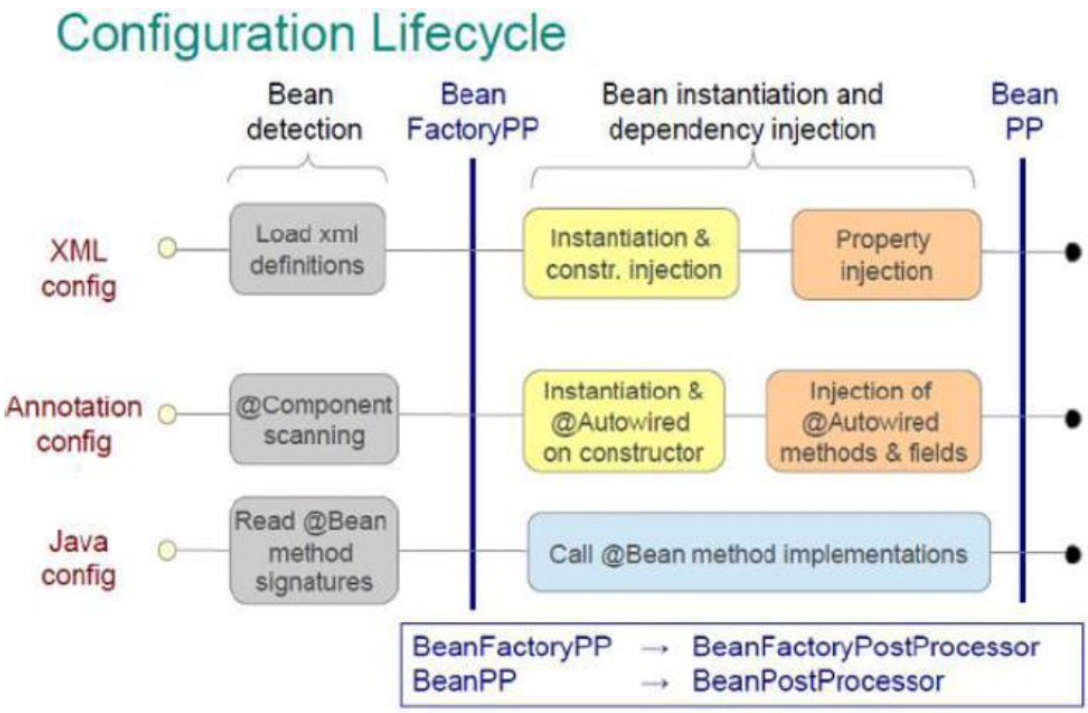


Figure 2: Zooming in on pre-instantiation

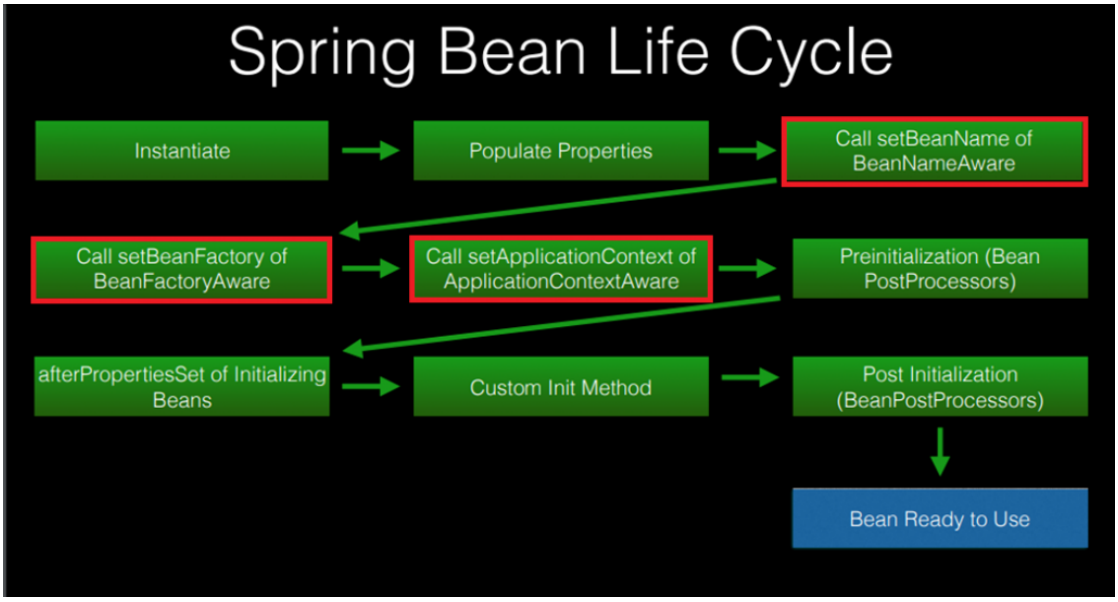


Figure 3: Zooming in on post-instantiation

2.1.2 Instantiate and run BeanFactoryPostProcessors

In a Spring application, a BeanFactoryPostProcessor can modify the definition of any bean. The BeanFactory object is passed as an argument to the postProcess() method of the BeanFactoryPostProcessor. BeanFactoryPostProcessor then works on the bean definitions or the configuration metadata of the bean before the beans are actually created. Spring provides several useful implementations of BeanFactoryPostProcessor, such as reading properties and registering a custom scope. We can write our own implementation of the BeanFactoryPostProcessor interface. To influence the order in which bean factory post processors are invoked, their bean definition methods may be annotated with the @Order annotation. If you are implementing your own bean factory post processor, the implementation class can also implement the Ordered interface.

2.1.3 Instantiate beans

Injects values and bean references into beans' properties.

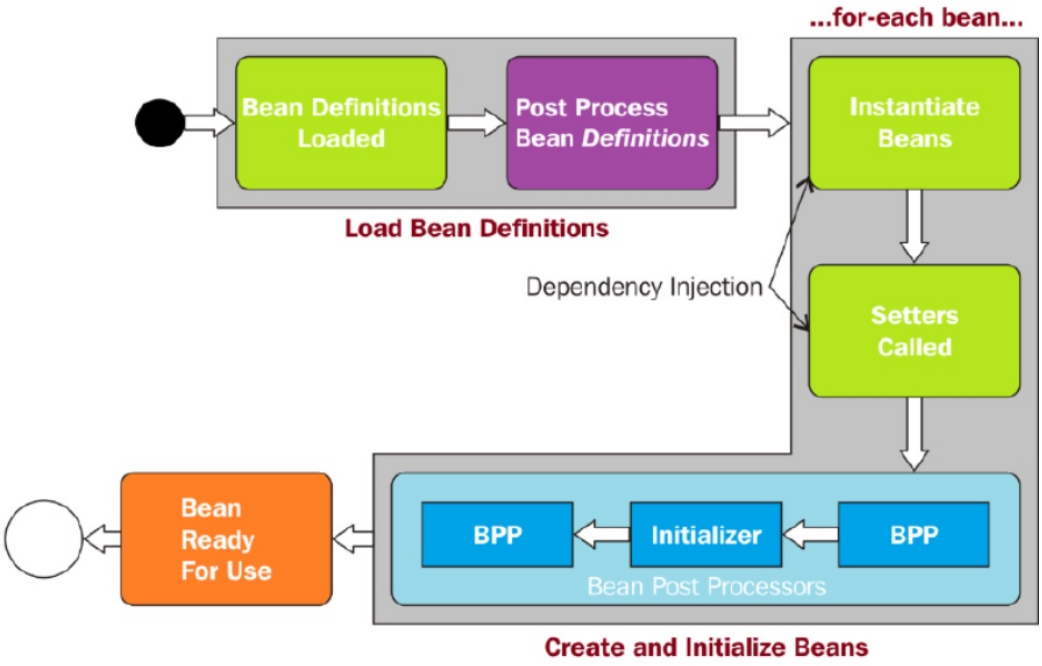


Figure 4:

2.1.4 Call BeanNameAware's setBeanName() for each bean implementing it

2.1.5 Call BeanFactoryAware's setBeanFactory() passing the bean factory for each bean implementing it

2.1.6 Call ApplicationContextAware's setApplicationContext for each bean implementing it

2.1.7 Before initialization: Run pre-initialization BeanPostProcessors

The Application context calls postProcessBeforeInitialization() for each bean implementing BeanPostProcessor.

```
1   public interface BeanPostProcessor {
2
3       /**
4       * Apply this {@code BeanPostProcessor} to the given
            new bean instance before any bean's
            initialization callbacks (like InitializingBean's
            afterPropertiesSet
5       * or a custom init-method).
6       **/
7       @Nullable
```

```
8         default Object
             postProcessBeforeInitialization(Object bean,
             String beanName) throws BeansException {
9             return bean;
10        }
11
12        /**
13        * Apply this {@code BeanPostProcessor} to the given
             new bean instance after any bean initialization
             callbacks (like InitializingBean's
             afterPropertiesSet
14        * or a custom init-method).
15        */
16        @Nullable
17        default Object postProcessAfterInitialization(Object
             bean, String beanName) throws BeansException {
18            return bean;
19        }
20    }
```

In postProcessBeforeInitialization and postProcessAfterInitialization, a bean implementing BeanPostProcessor can return anything it wants - even something completely different!

Figure 5 shows a no-op implementation.

2.1.8 Initialization: Call InitializingBean's afterPropertiesSet()

If a bean implements the InitializingBean interface, Spring calls its afterPropertiesSet() method. Used to initialize processes, load resources, etc. This approach is simple to use but it's not recommended because it will create tight coupling with the Spring framework in our bean implementations.

```
1 public interface InitializingBean {
2
3     /**
4     * Invoked by the containing BeanFactory after it has
         set all bean properties.
5     * This method allows the bean instance to perform
         validation of its overall configuration and final
         initialization when all bean properties have been
         set.
6     */
7     void afterPropertiesSet() throws Exception;
8
```

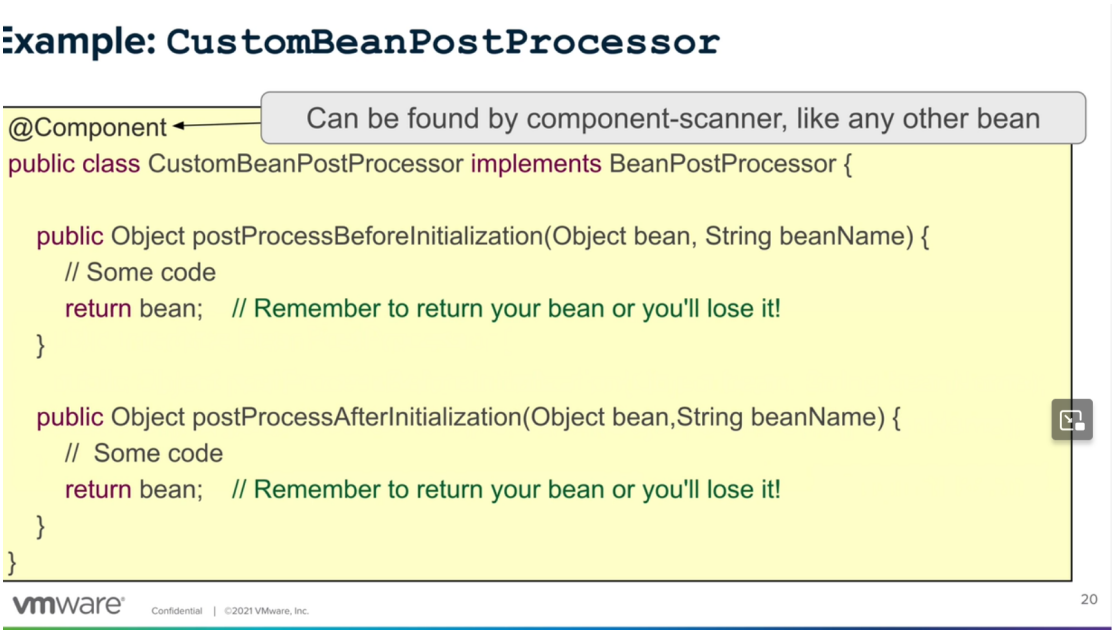


Figure 5: Custom bean postprocessor

```
9 }
```

2.1.9 Initialization: Init Method, @PostConstruct

Instead of implementing InitializingBean, you can use the init-method of the bean tag, the initMethod attribute of the @Bean annotation, and JSR 250's @PostConstruct annotation. Here we use the init-method attribute:

```
1     <bean name="myEmployeeService"
         class="com.journaldev.spring.service.MyEmployeeService"
2     init-method="init" destroy-method="destroy">
3     <property name="employee" ref="employee"></property>
4     </bean>
```

Using init-method is a solution when you don't own the class (and so, can't annotate it).

And here, the @PostConstruct annotation.

```
1     @PostConstruct
2     public void init(){
3         System.out.println("MyService init method called");
```

```
4     }
```

@PostConstruct and init-method are enabled by Spring's CommonAnnotationBeanPostProcessor. This is a BeanPostProcessor implementation that supports common Java annotations out of the box, in particular the JSR-250 annotations in the javax.annotation package.

It includes support for the javax.annotation.PostConstruct and javax.annotation.PreDestroy annotations - as init annotation and destroy annotation, respectively - through inheriting from InitDestroyAnnotationBeanPostProcessor with pre-configured annotation types.

```
1     public class CommonAnnotationBeanPostProcessor extends
         InitDestroyAnnotationBeanPostProcessor
2     implements InstantiationAwareBeanPostProcessor,
         BeanFactoryAware, Serializable {...}
```

2.1.10 After initialization: Run post-initialization BeanPostProcessors

The application context calls postProcessAfterInitialization() for each bean implementing BeanPostProcessor.

2.1.11 Bean ready to use

Your beans remain live in the application context until it is closed by calling the close() method of the application context.

2.1.12 Custom destruction

If a bean implements the DisposableBean interface, Spring calls its destroy() method to destroy any process or clean up the resources of your application. There are other methods to achieve this step-for example, you can use the destroy-method of the tag, the destroyMethod attribute of the '@Bean' annotation, and JSR 250's '@PreDestroy' annotation.

Precisely, the order of invocation is this:

- Any methods in the bean implementation class annotated with @PreDestroy are invoked.
- Any destroy method in a bean implementation class implementing the DisposableBean interface is invoked.
- Any custom bean destruction method is invoked. Bean destruction methods can be specified either in the value of the destroy-method attribute in the corresponding <bean >element in a Spring XML configuration or in the destroyMethod property of the @Bean annotation.

3 Dependency injection

Note: In addition to bean definitions that contain information on how to create a specific bean, the ApplicationContext implementations also permit the registration of existing objects that are created outside the container (by users).

This is done by accessing the ApplicationContext's BeanFactory through the `getBeanFactory()` method, which returns the `DefaultListableBeanFactory` implementation.

DefaultListableBeanFactory supports this registration through the `registerSingleton(..)` and `registerBeanDefinition(..)` methods.

3.1 Constructor-based

In the case of constructor-based dependency injection, the container will invoke a constructor with arguments each representing a dependency we want to set. This is the recommended way.

```
1         @Configuration
2         public class AppConfig {
3             @Bean
4             public Item item1() {
5                 return new ItemImpl1();
6             }
7             @Bean
8             public Store store() {
9                 return new Store(item1());
10            }
11        }
```

Resp.

```
1         <bean id="item1"
             class="org.baeldung.store.ItemImpl1" />
2         <bean id="store" class="org.baeldung.store.Store">
3         <constructor-arg type="ItemImpl1" index="0"
             name="item" ref="item1" />
4         </bean>
```

3.2 Method-based

For setter-based DI, the container will call setter methods of our class after invoking a no-argument constructor or no-argument static factory method to instantiate the bean.

```
1         @Bean
2         public Store store() {
```

```
3             Store store = new Store();
4             store.setItem(item1());
5             return store;
6         }
```

Resp.

```
1         <bean id="store" class="org.baeldung.store.Store">
2         <property name="item" ref="item1" />
3         </bean>
```

3.3 Field-based

In field-based DI, we can inject the dependencies by marking them with an @Autowired annotation. (This even works for private fields.) Field-based injection is not recommended - e.g., it makes testing harder.

```
1         public class Store {
2             @Autowired // deprecated
3             private Item item;
4         }
```

4 Application Context

The org.springframework.context.ApplicationContext interface represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on the components to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata can be represented as annotated component classes, configuration classes with factory methods, or external XML files or Groovy scripts.

Several implementations of the ApplicationContext interface are part of core Spring. In stand-alone applications, it is common to create an instance of AnnotationConfigApplicationContext or ClassPathXmlApplicationContext.

In most application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a plain web application scenario, a simple boilerplate web descriptor XML in the web.xml file of the application suffices (see Convenient ApplicationContext Instantiation for Web Applications). In a Spring Boot scenario, the application context is implicitly bootstrapped for you based on common setup conventions.

Implementing interfaces are:

```
1         // and abstract subclasses
2         AbstractApplicationContext
3
4         AnnotationConfigApplicationContext
5
6         AnnotationConfigWebApplicationContext
7
8         ClassPathXmlApplicationContext
9
10        FileSystemXmlApplicationContext
11
12        // incl. GenericGroovyApplicationContext,
             GenericWebApplicationContext,
             GenericXmlApplicationContext
13        GenericApplicationContext
14
15        GroovyWebApplicationContext
16
17        ResourceAdapterApplicationContext
18
19        StaticApplicationContext
20        StaticWebApplicationContext
21
22        XmlWebApplicationContext
```

4.1 Standard and Custom Events

Event handling in the ApplicationContext is provided through the ApplicationEvent class and the ApplicationListener interface. If a bean that implements the ApplicationListener interface is deployed into the context, every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified.

The event infrastructure offers an annotation-based model as well as the ability to publish any arbitrary event (that is, an object that does not necessarily extend from ApplicationEvent). When such an object is published, we wrap it in an event for you.

These are the standard events that Spring provides:

ContextRefreshedEvent Published when the ApplicationContext is initialized or refreshed (for example, by using the refresh() method on the ConfigurableApplicationContext interface). Here, "initialized" means that all beans are loaded, post-processor beans are detected and activated, singletons are pre-instantiated, and the ApplicationContext object is ready for use. As long as the context has not been closed, a refresh can be triggered multiple times, provided that the chosen ApplicationContext actually supports such "hot" refreshes. For example, XmlWebApplicationContext supports hot refreshes,

but GenericApplicationContext does not.

ContextStartedEvent Published when the ApplicationContext is started by using the start() method on the ConfigurableApplicationContext interface. Here, “started” means that all Lifecycle beans receive an explicit start signal. Typically, this signal is used to restart beans after an explicit stop, but it may also be used to start components that have not been configured for autostart (for example, components that have not already started on initialization).

ContextStoppedEvent Published when the ApplicationContext is stopped by using the stop() method on the ConfigurableApplicationContext interface. Here, “stopped” means that all Lifecycle beans receive an explicit stop signal. A stopped context may be restarted through a start() call.

ContextClosedEvent Published when the ApplicationContext is being closed by using the close() method on the ConfigurableApplicationContext interface or via a JVM shutdown hook. Here, ”closed” means that all singleton beans will be destroyed. Once the context is closed, it reaches its end of life and cannot be refreshed or restarted.

RequestHandledEvent A web-specific event telling all beans that an HTTP request has been serviced. This event is published after the request is complete. This event is only applicable to web applications that use Spring’s DispatcherServlet.

ServletRequestHandledEvent A subclass of RequestHandledEvent that adds Servlet-specific context information.

You can also create and publish your own custom events. The following example shows a simple class that extends Spring’s ApplicationEvent base class:

public class BlockedListEvent extends ApplicationEvent

private final String address; private final String content;

public BlockedListEvent(Object source, String address, String content) super(source); this.address = address; this.content = content;

// accessor and other methods...

To publish a custom ApplicationEvent, call the publishEvent() method on an ApplicationEventPublisher. Typically, this is done by creating a class that implements ApplicationEventPublisherAware and registering it as a Spring bean. The following example shows such a class:

Java

Kotlin

public class EmailService implements ApplicationEventPublisherAware

private List¡String¿ blockedList; private ApplicationEventPublisher publisher;

public void setBlockedList(List¡String¿ blockedList) this.blockedList = blockedList;

public void setApplicationEventPublisher(ApplicationEventPublisher publisher) this.publisher = publisher;

public void sendEmail(String address, String content) if (blockedList.contains(address)) publisher.publishEvent(new BlockedListEvent(this, address, content)); return; // send email...

At configuration time, the Spring container detects that EmailService implements ApplicationEventPublisherAware and automatically calls setApplicationEventPublisher().

In reality, the parameter passed in is the Spring container itself. You are interacting with the application context through its ApplicationEventPublisher interface.

To receive the custom ApplicationEvent, you can create a class that implements ApplicationListener and register it as a Spring bean. The following example shows such a class:

Java

Kotlin

public class BlockedListNotifier implements ApplicationListener¡BlockedListEvent¿

private String notificationAddress;

public void setNotificationAddress(String notificationAddress) this.notificationAddress = notificationAddress;

public void onApplicationEvent(BlockedListEvent event) // notify appropriate parties via notificationAddress...

Notice that ApplicationListener is generically parameterized with the type of your custom event (BlockedListEvent in the preceding example). This means that the onApplicationEvent() method can remain type-safe, avoiding any need for downcasting. You can register as many event listeners as you wish, but note that, by default, event listeners receive events synchronously. This means that the publishEvent() method blocks until all listeners have finished processing the event. One advantage of this synchronous and single-threaded approach is that, when a listener receives an event, it operates inside the transaction context of the publisher if a transaction context is available. If another strategy for event publication becomes necessary, for example, asynchronous event processing by default, see the javadoc for Spring's ApplicationEventMulticaster interface and SimpleApplicationEventMulticaster implementation for configuration options which can be applied to a custom "applicationEventMulticaster" bean definition. In these cases, ThreadLocals and logging context are not propagated for the event processing. See the @EventListener Observability section for more information on Observability concerns.

The following example shows the bean definitions used to register and configure each of the classes above:

¡bean id="emailService" class="example.EmailService"¿ ¡property name="blockedList"¿ ¡list¿ ¡value¿known.spammer@example.org¡/value¿ ¡value¿known.hacker@example.org¡/value¿ ¡value¿john.doe@example.org¡/value¿ ¡/list¿ ¡/property¿ ¡/bean¿

¡bean id="blockedListNotifier" class="example.BlockedListNotifier"¿ ¡property name="notificationAddr value="blockedlist@example.org"/¿ ¡/bean¿

¡!– optional: a custom ApplicationEventMulticaster definition –¿ ¡bean id="applicationEventMulticaster" class="org.springframework.context.event.SimpleApplicationEventMulticaster"¿ ¡property name="taskExecutor" ref="..."/¿ ¡property name="errorHandler" ref="..."/¿ ¡/bean¿

Putting it all together, when the sendEmail() method of the emailService bean is called, if there are any email messages that should be blocked, a custom event of type BlockedListEvent is published. The blockedListNotifier bean is registered as an ApplicationListener and receives the BlockedListEvent, at which point it can notify appropriate parties.

4.2 Annotation-based Event Listeners

You can register an event listener on any method of a managed bean by using the @EventListener annotation. Example:

```
1   public class BlockedListNotifier {
2
3       private String notificationAddress;
4
5       public void setNotificationAddress(String
           notificationAddress) {
6           this.notificationAddress = notificationAddress;
7       }
8
9       @EventListener
10      public void processBlockedListEvent(BlockedListEvent
           event) {
11          // notify appropriate parties via
               notificationAddress...
12      }
13  }
```

5 Configuration: Implicit vs. Explicit

Also referred to as Java-based (decoupled) and annotation-based.
with both types, bean naming works differently - see 8.

5.1 Java-based

Takes place completely in @Configuration classes. E.g.,

```
1       @Configuration
2       public class MyConfig {
3       @Bean
4       public AccountRepo AccountRepo(){}
5       }
```

5.2 Annotation-based

Bean definition and wiring take place completely in POJOs. For this to work, we need to enable component scanning.

```
1         @Configuration
2         @ComponentScan
3         public class MyConfig {}
4
5         @Component
6        public class AccountRepo {}
```

6 Annotations

6.1 Annotations for dependency injection

6.1.1 @Autowired

@Autowired marks a dependency which Spring is going to resolve and inject. We can use this annotation with constructor, setter, or field injection. E.g.,

```
1         class Car {
2             @Autowired
3             Engine engine;
4         }
```

Starting with version 4.3, we don't need to annotate constructors with @Autowired explicitly unless we declare at least two constructors.

@Autowired matches by type. If there are several classes matching the required type (e.g., implementing the same interface), @Autowired needs to be supplemented by @Qualifier:

```
1         @Component("Repo1")
2         class Repo1 implements Repo {}
3
4         @Component("Repo2")
5         class Repo2 implements Repo {}
6
7         @Component
8         public class Service1 implements ServiceX {
9             public Service1(@Qualifier("Repo2") Repo){}
10
11        }
```

If there is no @Qualifier given, @Autowired looks for a bean annotated with @Primary. If none exists, Spring will match by bean name (= bean id).

Here, Spring will look for a bean named x:

```
        // constructor injection
        @Autowired
        public MyBean(X x){}

       // method injection
       @Autowired
       public setX(X x){}

       // field injection
       @Autowired
       private X x;
```

6.1.2 Using @Primary in conjunction with @Qualified

For example, when you work with two different databases, using the @Primary annotation allows you to specify the main DataSource. Additional DataSources can be configured with different qualifiers, allowing you to inject them as needed.

```

    @Configuration
    public class DataSourceConfig {

        @Bean
        @Primary
        @ConfigurationProperties(prefix =
           "spring.datasource")
        public DataSource primaryDataSource() {
            return DataSourceBuilder.create().build();
        }

        @Bean
        @ConfigurationProperties(prefix =
           "spring.second-datasource")
        public DataSource secondaryDataSource() {
            return DataSourceBuilder.create().build();
        }
    }

\end{{lstlisting}

\subsubsection{@Bean}

```

```
23      @Bean marks a factory method which instantiates a
           Spring bean.
24
25      \begin{lstlisting}
26          @Bean
27          Engine engine() {
28              return new Engine();
29          }
```

Spring calls these methods when a new instance of the return type is required. All methods annotated with @Bean must be in @Configuration classes.

6.1.3 @Resource

The @Resource annotation matches by name, type, or qualifier (in this order). It is applicable to setter and field injection. Here's an example injecting a field. Note that the bean id and the corresponding reference attribute value must match:

```
1       @Configuration
2       public class MyAppContext {
3           @Bean(name="namedFile")
4           public File namedFile() {
5               File namedFile = new File("namedFile.txt");
6               return namedFile;
7           }
8       }
9
10      @ContextConfiguration(
11      loader=AnnotationConfigContextLoader.class,
12      classes= MyAppContext.class)
13      public class Xxx {
14          @Resource(name="namedFile")
15          private File defaultFile;
16      }
```

6.1.4 @Inject and @Named

Spring offers support for JSR-330 standard annotations (Dependency Injection). Those annotations are scanned in the same way as the Spring annotations. To use them, you need to have the relevant jars in your classpath.

The @Inject annotation matches by type, qualifier, or name (in this order). It is applicable to setter and field injection. With @Inject, the class reference variable's name and the bean name don't have to match.

To use the @Inject annotation, declare the javax.inject library as a Gradle or Maven dependency.

```
    public class MyAppContext {
        @Bean
        // no bean name specified - method name is used
        public File getSomeFile() {
            File namedFile = new File("namedFile.txt");
            return namedFile;
        }
    }

    @ContextConfiguration(
    loader=AnnotationConfigContextLoader.class,
    classes= MyAppContext.class)
    public class Xxx {
        @Inject
        private File defaultFile;
    }
```

Furthermore, you may declare your injection point as a Provider, allowing for on-demand access to beans of shorter scopes or lazy access to other beans through a Provider.get() call. The following example offers a variant of the preceding example:

```
    import jakarta.inject.Inject;
    import jakarta.inject.Provider;

    public class SimpleMovieLister {

        private Provider<MovieFinder> movieFinder;

        @Inject
        public void setMovieFinder(Provider<MovieFinder>
           movieFinder) {
            this.movieFinder = movieFinder;
        }

    public void listMovies() {
        this.movieFinder.get().findMovies(...);
        // ...
    }
}
```

If you would like to use a qualified name for the dependency that should be injected, you should use the `@Named` annotation, as the following example shows:

```
1     import jakarta.inject.Inject;
2     import jakarta.inject.Named;
3
4     public class SimpleMovieLister {
5
6         private MovieFinder movieFinder;
7
8         @Inject
9         public void setMovieFinder(@Named("main")
             MovieFinder movieFinder) {
10            this.movieFinder = movieFinder;
11        }
12
13        // ...
14    }
```

6.1.5 @Named and @ManagedBean: Standard Equivalents to the @Component Annotation

Instead of @Component, you can use @jakarta.inject.Named or jakarta.annotation.ManagedBean:

```
1
2 import jakarta.inject.Inject;
3 import jakarta.inject.Named;
4
5 @Named("movieListener")  // @ManagedBean("movieListener")
     could be used as well
6 public class SimpleMovieLister {
7
8     private MovieFinder movieFinder;
9
10    @Inject
11    public void setMovieFinder(MovieFinder movieFinder) {
12        this.movieFinder = movieFinder;
13    }
14
15    // ...
16 }
```

It is very common to use @Component without specifying a name for the component. @Named can be used in a similar fashion:

```
    import jakarta.inject.Inject;
    import jakarta.inject.Named;

    @Named
    public class SimpleMovieLister {

        private MovieFinder movieFinder;

        @Inject
        public void setMovieFinder(MovieFinder movieFinder)
           {
            this.movieFinder = movieFinder;
        }

        // ...
    }
```

When you use @Named or @ManagedBean, you can use component scanning in the exact same way as when you use Spring annotations:

```

@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
    // ...
}
```

I

6.1.6 @Value

We can use @Value for injecting property values into beans. It's compatible with constructor, setter, and field injection. E.g.,

```
        Engine(@Value("8") int cylinderCount) {
            this.cylinderCount = cylinderCount;
        }
```

This is an alternative to making explicit use of Spring's Environment bean. E.g.

```
1         public DataSource dataSource(
2          @Value("${db.driver}") String driver,
3          ...
4          )
5          }
```

6.1.7 @DependsOn

We can use this annotation to make Spring initialize other beans before the annotated one. Usually, this behavior is automatic, based on the explicit dependencies between beans. We only need this annotation when the dependencies are implicit, for example, JDBC driver loading or static variable initialization. E.g.,

```
1          @Bean
2          @DependsOn("fuel")
3          Engine engine() {
4              return new Engine();
5          }
```

6.1.8 @Lazy

This annotation behaves differently depending on where exactly we place it.

- In an @Bean-annotated bean factory method, it is used to delay the method call (hence the bean creation)
- With an @Configuration class, all contained @Bean methods will be affected
- For all other @Component classes, they will be initialized lazily when so annotated.
- @Autowired constructors, setters, and fields will be loaded lazily (via proxy).

```
1          @Configuration
2          @Lazy
3          class VehicleFactoryConfig {
4
5              @Bean
6              @Lazy(false)
7              Engine engine() {
8                  return new Engine();
9              }
10         }
```

6.1.9 @Scope

@Scope is used to define the scope of a @Component class or a @Bean definition. It can be either singleton, prototype, request, session, globalSession or some cust@Component.

6.2 Context Configuration Annotations

6.2.1 @Import

With @import, we can use specific @Configuration classes without component scanning.

```
1         @Import(VehiclePartSupplier.class)
2         class VehicleFactoryConfig {}
3
4         // use array to import several classes
5         @Import({Demo1.class, Demo2.class})
```

6.2.2 @ImportResource

We can import XML configurations with @ImportResource. We can specify the XML file locations with the locations argument, or with its alias, the value argument:

```
1         @Configuration
2         @ImportResource("classpath:/annotations.xml")
3         class VehicleFactoryConfig {}
```

6.2.3 @PropertySource

With this annotation, we define property *files* for application settings.

```
1         @Configuration
2         @PropertySource("classpath:/annotations.properties")
3         @PropertySource("classpath:/vehicle-factory.properties")
4         class VehicleFactoryConfig {}
```

These properties can be used by Spring's Environment bean, in addition to environment variables and Java system properties.

Allowed prefixes are classpath:, file:, and http:.

6.3 Bean annotations

6.3.1 @Profile

Profiles are a way to group bean definitions, for example:

- dev, test, prod environment
- jdbc, jpa [implementations]

The @Profile annotation may be used in any of the following ways:

- At class level in @Configuration classes.
- At class level in classes annotated with @Component or annotated with any other annotation that in turn is annotated with @Component.
- On methods annotated with the @Bean annotation.

The profile string may contain a simple profile name (for example, production) or a profile expression. A profile expression allows for more complicated profile logic to be expressed. The following operators are supported in profile expressions: `!,\&,|`.

To define alternative beans with different profile conditions, use distinct Java method names pointing to the same bean name via the @Bean name attribute:

```
1         @Bean("dataSource")
2         @Profile("development")
3         public DataSource standaloneDataSource(){
4
5         @Bean("dataSource")
6         @Profile("production")
7         public DataSource jndiDataSource() throws Exception
             {}
```

Spring uses two separate properties when determining which profiles are active, spring.profiles.active and spring.profiles.default:

- If spring.profiles.active is set, then its value determines which profiles are active.
- If spring.profiles.active isn't set, then Spring looks to spring.profiles.default.
- If neither spring.profiles.active nor spring.profiles.default is set, only those beans that aren't defined as being in a profile are created.

These properties can be set on the command line:

```
1         -Dspring.profiles.active=embedded.jpa
```

, programmatically:

```
1         System.setProperty("spring.profiles.active",
             "embedded.jpa");
```

, or via an annotation (@ActiveProfiles; integration tests only).

6.3.2 @ComponentScan

The @ComponentScan annotation is used together with @Configuration.

@ComponentScan can be used with and without arguments.

Without arguments, @ComponentScan tells Spring to scan the current package and all of its sub-packages.

With arguments, @ComponentScan tells which packages or classes to scan. E.g., specifying packages:

```
1 @Configuration
2 @ComponentScan(basePackages =
     "com.baeldung.annotations")
3 class VehicleFactoryConfig {}
```

Or else, specifying classes:

```
1 @Configuration
2 @ComponentScan(basePackageClasses =
     VehicleFactoryConfig.class)
3 class VehicleFactoryConfig {}
```

We can specify multiple package names, using spaces, commas, or semicolons as a separator.

```
1 @ComponentScan(basePackages =
     "springapp.animals;springapp.flowers")
2 @ComponentScan(basePackages =
     "animals,springapp.flowers")
3 @ComponentScan(basePackages = "springapp.animals
     springapp.flowers")
```

We could also apply a filter, choosing from a range of filter types. For example:

```
1 @ComponentScan(excludeFilters =
2 @ComponentScan.Filter(type=FilterType.REGEX,
3 pattern="com\\.baeldung\\.componentscan\\.springapp\\.flowers\\..*
```

Or:

```
1 @ComponentScan(excludeFilters =
2 @ComponentScan.Filter(type =
     FilterType.ASSIGNABLE\_TYPE, value = Rose.class))
```

6.3.3 @Component

@Component is a class-level annotation. During component scan, Spring automatically detects classes annotated with @Component.

```
1         @Component
2         class CarUtility {
3             // ...
4         }
```

@Repository, @Service, @Configuration, and @Controller are all meta-annotations of (i.e., themselves annotated with) @Component. E.g.,

```
1         @Component
2         public @interface Service {}
```

Spring also automatically picks them up during the component scanning process.

6.3.4 @Repository

```
1         @Repository
2         class VehicleRepository {
3             // ...
4         }
```

6.3.5 @Service

```
1         @Service
2         public class VehicleService {
3             // ...
4         }
```

6.3.6 @Controller

```
1         @Controller
2         public class VehicleController {
3             // ...
4         }
```

6.3.7 @Configuration

Configuration classes can contain bean definition methods annotated with @Bean.

```
@Configuration
class VehicleFactoryConfig {

    @Bean
    Engine engine() {
        return new Engine();
    }

}
```

6.4 Spring Boot Annotations

6.4.1 @SpringBootApplication

This is a combination of three annotations:

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
```

6.4.2 @ConfigurationProperties

Helps keep configuration clean (see 6).
This annotation has to be enabled via one of:

- @EnableConfigurationProperties on the application class

```
@SpringBootApplication
@EnableConfigurationProperties(
   ConnectionSettings.class)
public class App {
    // ...
}
```

- @ConfigurationPropertiesScan on the application class

```
@SpringBootApplication
@ConfigurationPropertiesScan
public class App {
```

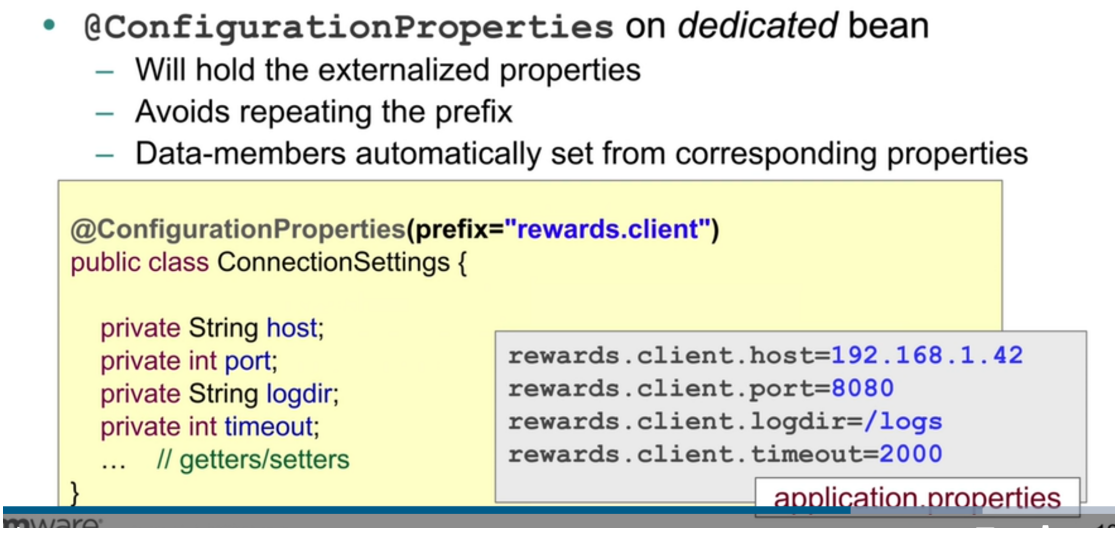


Figure 6:

```
4             // ...
5         }
```

- @Component on the configuration class

```
1         @Component
2         @ConfigurationProperties(prefix="...")
3         public class ConnectionSettings {
4             // ...
5         }
```

6.4.3 @ConditionalOnX

Determine what auto configuration does. For example: @ConditionalOnBean, @ConditionalOnMissingBean, @ConditionalOnClass, @ConditionalOnMissingClass, @ConditionalOnProperty.

For example, @Profile is such a condition.

6.4.4 RestController

Includes @Controller and @ResponseBody.

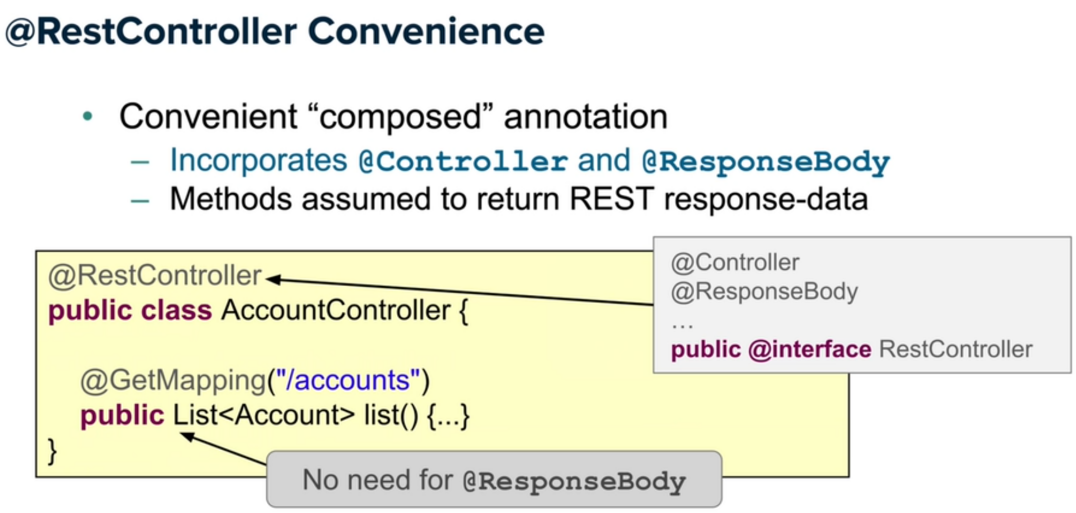


Figure 7: RestController convenience annotation.

6.4.5 Request URI Decomposition: @RequestParam, @PathVariable

Do implicit type conversion of arguments.

```
1     // localhost:8080/account?userid=12345
2     @GetMapping("/account")
3     public List<Account> list(@RequestParam("userid"} int
         userid) {}
4
5     // localhost:8080/accounts/12345
6     @GetMapping("/accounts/{accountId}")
7     public Account find (@PathVariable("accountId"} long
         id) {}
8
9     // if argument name is missing, will take from the
         mapping
10    // could also have
11
12    // localhost:8080/account?overdrawn=12345
13    @GetMapping("/account")
14    public List<Account> list(@RequestParam int overdrawn)
         {}
15
16    // localhost:8080/accounts/12345
17    @GetMapping("/accounts/{accountId}")
18    public Account find (@PathVariable long id) {}
19
```

```
20      // localhost:8080/accounts/12345?overdrawn=true
21      @GetMapping("/accounts/{accountId}")
22      public Account find (
23          @PathVariable long accountId,
24          @RequestParam boolean overdrawn
25      ) {}
```

6.4.6 @ResponseBody

Causes Java objects returned by the Controller to be processed by HttpMessageConverters in order to return information to the client in the form requested in the Accept header.

Example (assuming the client requests JSON):

Here, the @ResponseBody annotation tells a controller that the *object* returned is automatically serialized into *JSON* and passed back into the *HttpResponse* object.

Suppose we have a custom Response object:

```
1     public class ResponseTransfer {
2         private String text;
3
4         // standard getters/setters
5     }
```

Next, the associated controller can be implemented:

```
1     @Controller
2     @RequestMapping("/post")
3     public class ExamplePostController {
4
5         @Autowired
6         ExampleService exampleService;
7
8         @PostMapping("/response")
9         @ResponseBody
10        public ResponseTransfer postResponseController(
11        @RequestBody LoginForm loginForm) {
12            return new ResponseTransfer("Thanks For
                  Posting!!!");
13        }
14    }
```

6.4.7 @ResponseStatus

Used to return a status other than 200.

```
1         @ResponseStatus(HttpStatus.NO_CONTENT)
2         public void updateOrder(...){}
```

6.4.8 @RequestBody

Used to extract the request body.

More precisely, @RequestBody maps the HttpRequest body to a transfer or domain object, enabling automatic deserialization of the inbound HttpRequest body onto a Java object.

Example:

```
1
2     @PostMapping("/request")
3     public ResponseEntity postController(
4     @RequestBody LoginForm loginForm) {
5
6         exampleService.fakeAuthenticate(loginForm);
7         return ResponseEntity.ok(HttpStatus.OK);
8     }
```

Here, Spring automatically deserializes the JSON into a Java type, assuming an appropriate one is specified.

```
1         @ResponseStatus(HttpStatus.NO_CONTENT)
2         public void updateOrder(...){}
```


```
1         @ResponseStatus(HttpStatus.NO_CONTENT)
2         public void updateOrder(...){}
```


7 Aware Interfaces

Indicates that the bean is eligible to be notified by the Spring container through the callback methods. A typical use case for BeanNameAware could be acquiring the bean name for logging or wiring purposes. For the BeanFactoryAware it could be the ability to use a spring bean from legacy code. In most cases, we should avoid using any of the Aware interfaces, unless we need them. Implementing these interfaces will couple the code to the Spring framework.

7.1 BeanNameAware

Makes the object aware of the bean name defined in the container.

```
1     public class MyBeanName implements BeanNameAware {
2         @Override
3         public void setBeanName(String beanName) {
4             System.out.println(beanName);
5         }
6     }
7     @Configuration
8     public class Config {
9         @Bean(name = "myCustomBeanName")
10        public MyBeanName getMyBeanName() {
11            return new MyBeanName();
12        }
13    }
14    AnnotationConfigApplicationContext context
15    = new AnnotationConfigApplicationContext(Config.class);
16    MyBeanName myBeanName =
         context.getBean(MyBeanName.class);
```

7.2 BeanFactoryAware

Provides access to the BeanFactory which created the object.

```
1         public class MyBeanFactory implements
             BeanFactoryAware {
2             private BeanFactory beanFactory;
3             @Override
4             public void setBeanFactory(BeanFactory
                 beanFactory) throws BeansException {
5                 this.beanFactory = beanFactory;
6             }
7             public void getMyBeanName() {
8                 MyBeanName myBeanName =
                     beanFactory.getBean(MyBeanName.class);
9                 System.out.println(beanFactory.isSingleton("myCustomBeanNa
10            }
11        }
12        MyBeanFactory myBeanFactory =
             context.getBean(MyBeanFactory.class);
13        myBeanFactory.getMyBeanName();}
```

7.3 ApplicationContextAware

```
1  public class ApplicationContextAwareImpl implements
       ApplicationContextAware {
2      @Override
3      public void
           setApplicationContext(ApplicationContext
           applicationContext) throws BeansException {
4          User user = (User)
               applicationContext.getBean("user");
5          System.out.println("User Id: " +
               user.getUserId() + " User Name :" +
               user.getName());}}
```

8 Bean Naming

8.1 Default Bean Naming

8.1.1 Class-level ("Annotation-based configuration")

For an annotation used at the class level (@Component, @Service, @Controller), Spring uses the class name and converts the first letter to lowercase. Custom names may be configured in the annotation's value attribute.

The type is determined from the annotated class, typically resulting in the actual implementation class.

```
1  @Service
2  public class LoggingService { // bean name =
       loggingService
3
4  }
```

8.1.2 Method-level ("Java configuration")

When in a @Configuration class we use the @Bean annotation on a method, Spring uses the method name for the bean name.

```
1  @Configuration
2  public class AuditConfiguration {
3      @Bean
4      public AuditService audit() {
5          return new AuditService();
```

```
6               }
7           }
```

8.2 Custom naming

```
1           @Component("myBean")
2           public class MyCustomComponent {
3           }
```

Custom names may be configured in @Bean's value attribute.
The type is determined from the method return type, typically resulting in an interface.

8.3 Naming Beans With @Bean and @Qualifier

8.3.1 @Bean With Value

The @Bean annotation is applied at the method level, and by default, Spring uses the method name as a bean name. We can override this using the @Bean annotation.

```
1           @Configuration
2           public class MyConfiguration {
3               @Bean("beanComponent")
4               public MyCustomComponent myComponent() {
5                   return new MyCustomComponent();
6               }
7           }
```

8.3.2 @Qualifier With Value

We can also use the @Qualifier annotation to name the bean.

```
1           @Component
2           @Qualifier("cat")
3           public class Cat implements Animal {
4               @Override
5               public String name() {
6                   return "Cat";
7               }
8           }
9           @Component
10          @Qualifier("dog")
```

```
11          public class Dog implements Animal {
12              @Override
13              public String name() {
14                  return "Dog";
15              }
16          }
17          @Service
18          public class PetShow {
19              private final Animal dog;
20              private final Animal cat;
21
22              public PetShow (@Qualifier("dog")Animal dog,
                   @Qualifier("cat")Animal cat) {
23                  this.dog = dog;
24                  this.cat = cat;
25              }
26              public Animal getDog() {
27                  return dog;
28              }
29              public Animal getCat() {
30                  return cat;
31              }
32          }
```

9 Spring Expression Language vs. Property Evaluation

Expressions in @Value annotations are of two types:

- Expressions starting with $. Such expressions reference a property name in the application's environment. These expressions are evaluated by the PropertySourcesPlaceholderConfigurer BeanFactoryPostProcessor prior to bean creation and can only be used in @Value annnotations.
- Expressions starting with #. These expressions are parsed by a SpEL expression parser, and are evaluated by a SpEL expression instance.

In some cases, both can be used. For example, property values by default are Strings, but may be converted to primitives implicitly. So, both of these work:

```
1           @Value("${daily.limit}")
2           int limit;
3
4           @Value("#{environment['daily.limit']}")
```

```
5         int limit;
```

But if computations are to be performed, or object types are required, SpEL has to be used:

```
1         // NO
2         @Value("${daily.limit} * 2")
3
4         // instead, do
5         @Value("#{new Integer(environment['daily.limit']})
             * 2")
```

To provide defaults, use a colon with property evaluation, and ?: in SpEL.

```
1         @Value("${daily.limit}: 1000")
2         int limit;
3
4         @Value("#{environment['daily.limit']} ?: 1000")
5         int limit;
```

In addition to application-defined beans, SpEL can make use of beans implicitly provided by Spring, namely environment, systemProperties, and systemEnvironment.

10 AOP in Spring

10.1 Core AOP Concepts

10.1.1 Join Point

A point during the execution of a program, such as the execution of a method or the handling of an exception.

In Spring AOP, a join point always represents a method execution.

10.1.2 Point Cut

An expression that selects one or more join points.

Although Spring supports various AspectJ pointcut designators, the most commonly used one is `execution`.

For this designator, the syntax pattern is as follows:

```
1     execution(
2     modifiers-pattern?
3     ret-type-pattern
4     declaring-type-pattern.?name-pattern(param-pattern)
```

```
5       throws-pattern?
6       )
```

All parts except the returning type pattern (ret-type-pattern in the preceding snippet), the name pattern, and the parameters pattern are optional.

- The returning type pattern determines what the return type of the method must be in order for a join point to be matched. * is most frequently used as the returning type pattern. It matches any return type. A fully-qualified type name matches only when the method returns the given type.
- The name pattern matches the method name. You can use the * wildcard as all or part of a name pattern. If you specify a declaring type pattern, include a trailing . to join it to the name pattern component.
- The parameters pattern is slightly more complex: () matches a method that takes no parameters, whereas (..) matches any number (zero or more) of parameters. The (*) pattern matches a method that takes one parameter of any type. (*,String) matches a method that takes two parameters. The first can be of any type, while the second must be a String.

Examples:

```
1  // The execution of any public method:
2  execution(public * *(..))
3
4  // The execution of any method with a name that begins with
       set:
5  execution(* set*(..))
6
7  // The execution of any method defined by the
       AccountService interface:
8  execution(* com.xyz.service.AccountService.*(..))
9
10 // The execution of any method defined in the service
       package:
11 execution(* com.xyz.service.*.*(..))
12
13 //The execution of any method defined in the service
       package or one of its sub-packages:
14 execution(* com.xyz.service..*.*(..))
15
16 // There is one directory between rewards and restaurant.
17 execution(* rewards.*.restaurant.*.*(..))
18
```

```
19 // There are 0 or more directories between rewards and
       restaurant.
20 execution(* rewards..restaurant.*.*(..))
21
22 // There must be at least 1 directory before restaurant.
23 // omitting the star is not allowed
24 execution(* *..restaurant.*.*(..))
25
26 // Any join point (method execution only in Spring AOP)
       within the service package:
27 within(com.xyz.service.*)
28
29 // Any join point (method execution only in Spring AOP)
       within the service package or one of its sub-packages:
30 within(com.xyz.service..*)
31
32 // Any join point (method execution only in Spring AOP)
       where the proxy implements the AccountService interface:
33 this(com.xyz.service.AccountService)
34
35 // Any join point (method execution only in Spring AOP)
       where the target object implements the AccountService
       interface:
36 target(com.xyz.service.AccountService)
37
38 // Any join point (method execution only in Spring AOP)
       that takes a single parameter and where the argument
       passed at runtime is Serializable.
39 // Note that the pointcut given in this example is
       different from execution(* *(java.io.Serializable)). The
       args version matches if the argument passed at runtime
       is Serializable, and the execution version matches if
       the method signature declares a single parameter of type
       Serializable.
40 args(java.io.Serializable)
41
42 // Any join point (method execution only in Spring AOP)
       where the target object has a @Transactional annotation:
43 @target(org.springframework.transaction.annotation.Transactional)
44
45 // Any join point (method execution only in Spring AOP)
       where the declared type of the target object has an
       @Transactional annotation:
46 @within(org.springframework.transaction.annotation.Transactional)
```

```
47
48 // Any join point (method execution only in Spring AOP)
       where the executing method has an @Transactional
       annotation:
49 @annotation(org.springframework.transaction.annotation.Transactional)
50
51 // Any join point (method execution only in Spring AOP)
       which takes a single parameter, and where the runtime
       type of the argument passed has the @Classified
       annotation:
52 @args(com.xyz.security.Classified)
53
54 // Any join point (method execution only in Spring AOP) on
       a Spring bean named tradeService:
55 bean(tradeService)
56
57 // Any join point (method execution only in Spring AOP) on
       Spring beans having names that match the wildcard
       expression *Service:
58 bean(*Service)
```

10.1.3 Advice

Code to be executed at a particular join point. Types:

- Before-advice is executed before calling the target method.

```
1         @Before("execution(void set*(*))
```

- After-advice is executed after the target method, whatever its outcome.

```
1         @Before("execution(void set*(*))
```

- After-returning: executed after the target returns successfully. This advice will never execute if the target throws any exception. The return parameter also gives access to the returned object.

```
1         @AfterReturning(value="execution(*
              service..*(..))", return="reward")
2         public void audit(Join Point jp, Reward reward)
              {
```

```
3             auditService.logEvent(jp.getSignature() +
                 ": " + reward.toString());
4         }
```

- After-throwing: executed after the target throws an exception. Also gives access to the exception.

```
1         // Repositories in any package
2         @AfterThrowing(value="execution(*
             *..Repository.*(..))", throwing="e")
3         // also have to match the type of the exception
4         public void report(JoinPoint jp,
             DataAccessException e) {
5         mailService.mailFailure(jp.getSignature(), e);
6         }
```

While this advice cannot prevent an exception to be thrown, it can throw a more user-friendly exception instead:

```
1         @AfterThrowing(value="execution(*
             *..Repository.*(..))", throwing"e")
2         public void report(JoinPoint jp,
             DataAccessException e) {
3             mailService.mailFailure(jp.getSignature(),
                 e);
4             throw new RewardsException();
5         }
```

- Around: executed two times, before and after invocation of the target method. Must call proceed() to delegate to the target. See 8.

10.1.4 Aspect

The combination of point cut and advice. The @aspect annotation needs to be explicitly enabled by @EnableAspectJConfiguration set in the context (Config) class.

This will cause an extension of AbstractAutoProxyCreator to run, a BeanPostProcessor that wraps a bean with an AOP proxy. See 9.

An aspect can get context information by injecting the JoinPoint into the advice. See fig. 10.

```
1     public abstract class AbstractAutoProxyCreator extends
         ProxyProcessorSupport
```

- Use *@Around* annotation *and* a *ProceedingJoinPoint*
  - Inherits from *JoinPoint* and adds the *proceed()* method

```
@Around("execution(@example.Cacheable * rewards.service..*.*(..))")
public Object cache(ProceedingJoinPoint point)  throws Throwable {
   Object value = cacheStore.get(CacheUtils.toKey(point));

   if (value != null)  return value;

   value = point.proceed();
   cacheStore.put(CacheUtils.toKey(point), value);
   return value;
}
```

Value exists? If so just return it

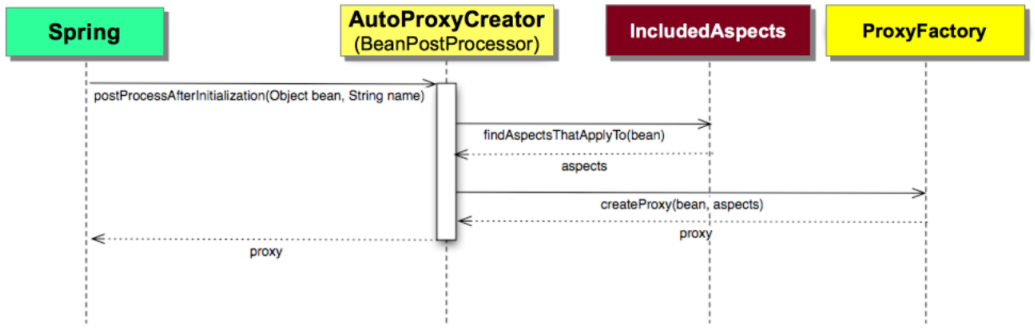
Proceed *only* if not already cached

Cache values returned by *cacheable* services

vmware

47

Figure 8: Around Advice



This following shows the internal structure of a created proxy and what happens when it is invoked:

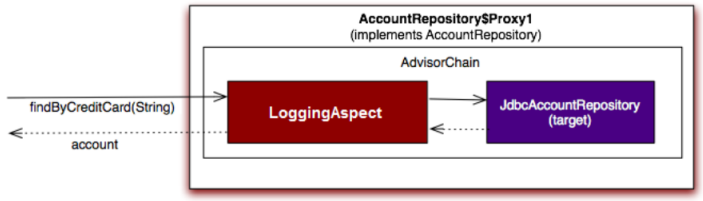


Figure 9: Proxy Creation.

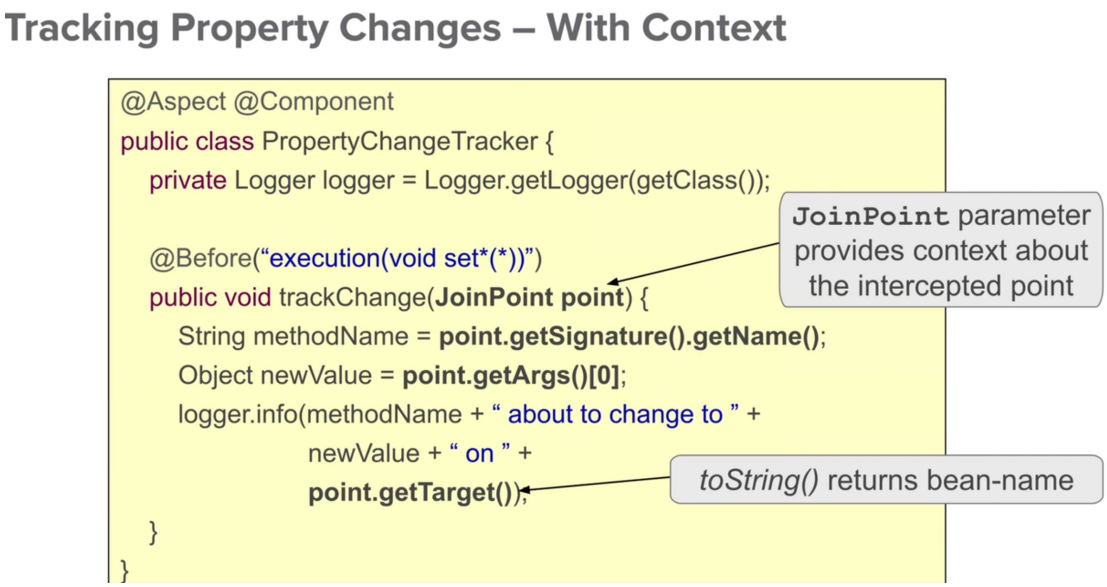


Figure 10: Automatic JoinPoint injection

```
2     implements SmartInstantiationAwareBeanPostProcessor,
         BeanFactoryAware {
3        //...
4
5        @Override
6        public Object
            postProcessBeforeInstantiation(Class<?>
            beanClass, String beanName) {
7           Object cacheKey = getCacheKey(beanClass,
               beanName);
8
9           if (!StringUtils.hasLength(beanName) ||
               !this.targetSourcedBeans.contains(beanName))
               {
10             if
                  (this.advisedBeans.containsKey(cacheKey))
                  {
11                return null;
12             }
13             if (isInfrastructureClass(beanClass) ||
                  shouldSkip(beanClass, beanName)) {
14                this.advisedBeans.put(cacheKey,
                     Boolean.FALSE);
```

```
15                      return null;
16                  }
17              }
18          }
19
20          @Override
21          public Object
               postProcessAfterInitialization(@Nullable Object
               bean, String beanName) {
22              if (bean != null) {
23                  Object cacheKey =
                       getCacheKey(bean.getClass(), beanName);
24                  if
                       (this.earlyProxyReferences.remove(cacheKey)
                       != bean) {
25                      return wrapIfNecessary(bean, beanName,
                           cacheKey);
26                  }
27              }
28              return bean;
29          }
30      }
```

10.1.5 More Terminology

Introduction Declaring additional methods or fields on behalf of a type. Spring AOP lets you introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an IsModified interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)

Target object An object being advised by one or more aspects. Also referred to as the "advised object". Since Spring AOP is implemented by using runtime proxies, this object is always a proxied object.

AOP proxy An object created by the AOP framework in order to implement the aspect contracts (advice method executions and so on). In the Spring Framework, an AOP proxy is a JDK dynamic proxy or a CGLIB proxy.

Weaving Linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. *Spring AOP, like other pure Java AOP frameworks*, performs weaving at *runtime*.

10.2 AOP Proxies

Spring AOP defaults to using standard *JDK dynamic proxies* for AOP proxies. This enables any interface (or set of interfaces) to be proxied.

Spring AOP can also use CGLIB proxies. This is necessary to proxy classes rather than interfaces. By default, CGLIB is used if a business object does not implement an interface.

If the target object to be proxied implements at least one interface, a JDK dynamic proxy is used, and all of the interfaces implemented by the target type are proxied. If the target object does not implement any interfaces, a CGLIB proxy is created which is a runtime-generated subclass of the target type.

10.3 Implications of Using a Proxy

Here, we create an object instance that calls a method on itself (using this).

```
1     public class SimplePojo implements Pojo {
2
3         public void foo() {
4             // this next method invocation is a direct call
                 on the 'this' reference
5             this.bar();
6         }
7
8         public void bar() {
9             // some logic...
10        }
11    }
12
13    public class Main {
14
15        public static void main(String[] args) {
16            Pojo pojo = new SimplePojo();
17            // this is a direct method call on the 'pojo'
                 reference
18            pojo.foo();
19        }
20    }
```

When SimplePojo is proxied, the same call will not result in bar() being intercepted, since bar() is not called on the proxy, but the `this` reference the object has to itself.

```
1     public class Main {
2
3         public static void main(String[] args) {
```

```
4             ProxyFactory factory = new ProxyFactory(new
                 SimplePojo());
5             factory.addInterface(Pojo.class);
6             factory.addAdvice(new RetryAdvice());
7
8             Pojo pojo = (Pojo) factory.getProxy();
9             // this is a method call on the proxy!
10            pojo.foo();
11        }
12    }
```

10.4 Programmatic Creation of @AspectJ Proxies

In addition to declaring aspects in your xml configuration by using either aop:config or aop:aspectj-autoproxy, it is also possible to programmatically create proxies that advise target objects.

You can use the org.springframework.aop.aspectj.annotation.AspectJProxyFactory class to create a proxy for a target object that is advised by one or more @AspectJ aspects.

```
1     // create a factory that can generate a proxy for the
         given target object
2     AspectJProxyFactory factory = new
         AspectJProxyFactory(targetObject);
3
4     // add an aspect, the class must be an @AspectJ aspect
5     // you can call this as many times as you need with
         different aspects
6     factory.addAspect(SecurityManager.class);
7
8     // you can also add existing aspect instances, the type
         of the object supplied
9     // must be an @AspectJ aspect
10    factory.addAspect(usageTracker);
11
12    // now get the proxy object...
13    MyInterfaceType proxy = factory.getProxy();
```

11 Transaction Management

11.1 Declarative

The Spring Framework's declarative transaction management is made possible with Spring aspect-oriented programming (AOP). Due to reliance on AOP, we can customize transactional behavior - for example, insert custom behavior in the case of transaction rollback, or add arbitrary advice, along with transactional advice.

The concept of rollback rules is important. They let you specify which exceptions (and throwables) should cause automatic rollback. You can specify this declaratively, in configuration, not in Java code. So, although you can still call `setRollbackOnly()` on the TransactionStatus object to roll back the current transaction, most often you can specify a rule that MyApplicationException must always result in rollback.

The combination of AOP with transactional metadata yields an AOP proxy that uses a `TransactionInterceptor` in conjunction with an appropriate `TransactionManager` implementation to drive transactions around method invocations.

The following image shows a conceptual view of calling a method on a transactional proxy:

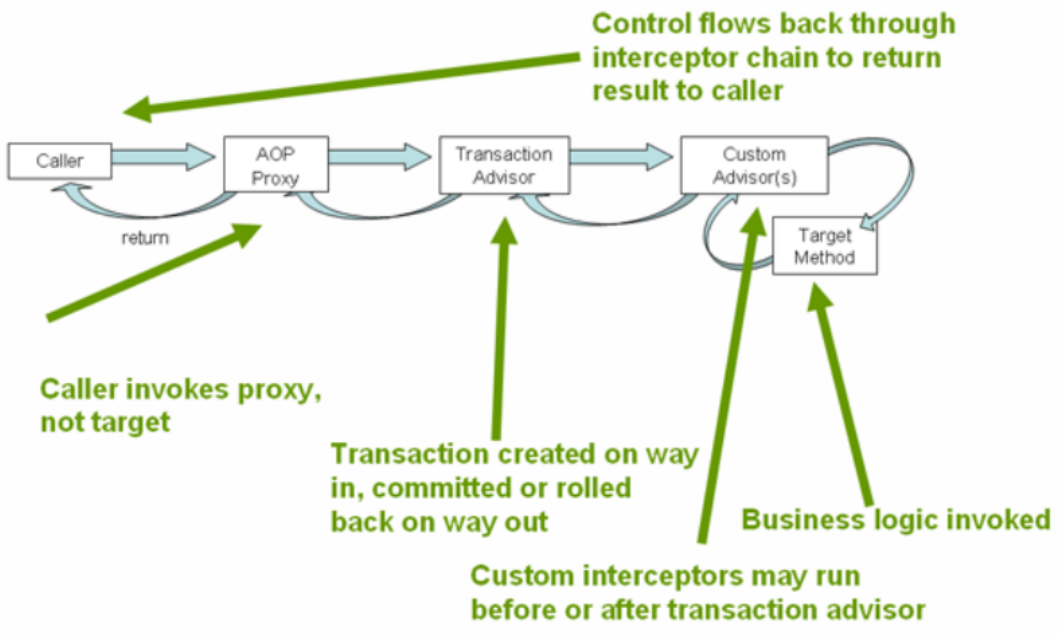


Figure 11: Spring Transaction Management Overview

Since declarative transaction management is made possible using Spring AOP, which by default uses JDK Dynamic Proxies, only public methods called from outside the class are affected by the @Transactional annotation.

11.2 Manual

Instead of using the @Transactional annotation, transactions can be managed programmatically using TransactionTemplate.

To be used, it needs to be initialized it with a PlatformTransactionManager.

Example:

```
1   class ManualTransactionIntegrationTest {
2
3       @Autowired
4       private PlatformTransactionManager
           transactionManager;
5
6       private TransactionTemplate transactionTemplate;
7
8       @BeforeEach
9       void setUp() {
10          transactionTemplate = new
               TransactionTemplate(transactionManager);
11      }
12
13      // omitted
14  }
```

When using Spring Boot, an appropriate bean of type PlatformTransactionManager will be automatically registered, so we just need to inject it. Otherwise, we have to manually register a PlatformTransactionManager bean.

The correct order of operations when using TransactionTemplate in programmatic transaction management is:

- begin transaction
- execute callback (which contains the transactional code)
- commit transaction if the callback executes successfully
- rollback transaction if an exception occurs during callback execution

12 Data Access with JDBC

The following table shows which actions Spring takes care of and which actions are your responsibility.

Action	Spring	You
Define connection parameters		x
Open the connection.	x	
Specify the SQL statement.		x
Declare parameters and provide parameter values		x
Prepare and run the statement.	x	
Set up the loop to iterate through the results (if any).	x	
Do the work for each iteration.		x
Process any exception.	x	
Handle transactions.	x	
Close the connection, the statement, and the resultset.	x	

12.1 Package Hierarchy

The Spring Framework's JDBC abstraction framework consists of four different packages:

- core: The org.springframework.jdbc.core package contains the JdbcTemplate class and its various callback interfaces, plus a variety of related classes. A subpackage named org.springframework.jdbc.core.simple contains the SimpleJdbcInsert and SimpleJdbcCall classes. Another subpackage named org.springframework.jdbc.core.namedparam contains the NamedParameterJdbcTemplate class and the related support classes. See Using the JDBC Core Classes to Control Basic JDBC Processing and Error Handling, JDBC Batch Operations, and Simplifying JDBC Operations with the SimpleJdbc Classes.

- datasource: The org.springframework.jdbc.datasource package contains a utility class for easy DataSource access and various simple DataSource implementations that you can use for testing and running unmodified JDBC code outside of a Jakarta EE container. A subpackage named org.springframework.jdbc.datasource.embedded provides support for creating embedded databases by using Java database engines, such as HSQL, H2, and Derby. See Controlling Database Connections and Embedded Database Support.

- object: The org.springframework.jdbc.object package contains classes that represent RDBMS queries, updates, and stored procedures as thread-safe, reusable objects. See Modeling JDBC Operations as Java Objects. This style results in a more object-oriented approach, although objects returned by queries are naturally disconnected from the database. This higher-level of JDBC abstraction depends on the lower-level abstraction in the org.springframework.jdbc.core package.

- support: The org.springframework.jdbc.support package provides SQLException translation functionality and some utility classes. Exceptions thrown during JDBC processing are translated to exceptions defined in the org.springframework.dao package. This means that code using the Spring JDBC abstraction layer does not need to implement JDBC or RDBMS-specific error handling. All translated

exceptions are unchecked, which gives you the option of catching the exceptions from which you can recover while letting other exceptions be propagated to the caller. See Using SQLExceptionTranslator.

12.2 Using JdbcTemplate

12.2.1 Queries

Examples:

```
1
2     Actor actor = jdbcTemplate.queryForObject(
3     "select first_name, last_name from t_actor where id =
         ?",
4     (resultSet, rowNum) -> {
5         Actor newActor = new Actor();
6         newActor.setFirstName(
7             resultSet.getString("first_name"));
8         newActor.setLastName(
9             resultSet.getString("last_name"));
10        return newActor;
11    },
12    1212L);
13
14    List<Actor> actors = this.jdbcTemplate.query(
15    "select first_name, last_name from t_actor",
16    (resultSet, rowNum) -> {
17        Actor actor = new Actor();
18        actor.setFirstName(
19            resultSet.getString("first_name"));
20        actor.setLastName(
21            resultSet.getString("last_name"));
22        return actor;
23    });
24
25    // If the last two snippets of code actually existed in
         the same application, it would make sense to remove
         the duplication present in the two RowMapper lambda
         expressions and extract them out into a single field
         that could then be referenced by DAO methods as
         needed. For example, it may be better to write the
         preceding code snippet as follows:
26
27    private final RowMapper<Actor> actorRowMapper =
         (resultSet, rowNum) -> {
```

```
28          Actor actor = new Actor();
29          actor.setFirstName(
30              resultSet.getString("first_name"));
31          actor.setLastName(
32              resultSet.getString("last_name"));
33          return actor;
34      };
35
36      public List<Actor> findAllActors() {
37          return this.jdbcTemplate.query("select first_name,
                last_name from t_actor", actorRowMapper);
38      }
```

12.2.2 Updating (INSERT, UPDATE, and DELETE) with JdbcTemplate

You can use the update(..) method to perform insert, update, and delete operations. Parameter values are usually provided as variable arguments or, alternatively, as an object array.

```
1       //The following example inserts a new entry:
2       this.jdbcTemplate.update(
3       "insert into t_actor (first_name, last_name) values (?,
            ?)",
4       "Leonor", "Watling");
5
6       //The following example updates an existing entry:
7       this.jdbcTemplate.update(
8       "update t_actor set last_name = ? where id = ?",
9       "Banjo", 5276L);
10
11      //The following example deletes an entry:
12      this.jdbcTemplate.update(
13      "delete from t_actor where id = ?",
14      Long.valueOf(actorId));
```

12.2.3 Other JdbcTemplate Operations

You can use the execute(..) method to run any arbitrary SQL. Consequently, the method is often used for DDL statements. It is heavily overloaded with variants that take callback interfaces, binding variable arrays, and so on.

```
1       //The following example creates a table:
```

```
2     this.jdbcTemplate.execute("create table mytable (id
          integer, name varchar(100))");
3
4     //The following example invokes a stored procedure:
5     this.jdbcTemplate.update(
6     "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",
7     Long.valueOf(unionId));
```

12.2.4 Callback interfaces

JdbcTemplate callback interfaces (functional interfaces; implementors must implement the abstract method):

RowCallbackHandler	processRow(ResultSet rs)	process each row
ResultSetExtractor <T >	extractData(ResultSet rs)	process entire ResultSet
RowMapper <T >	mapRow(ResultSet rs, int rowNum)	map each row

13 Data Access with JPA

13.1 Repository Query Language

Example (see https://docs.spring.io/spring-data/commons/reference/repositories/query-methods-details.html):

```
1    interface PersonRepository extends Repository<Person,
        Long> {
2
3       List<Person>
           findByEmailAddressAndLastname(EmailAddress
           emailAddress, String lastname);
4
5       // Enables the distinct flag for the query
6       List<Person>
           findDistinctPeopleByLastnameOrFirstname(String
           lastname, String firstname);
7       List<Person>
           findPeopleDistinctByLastnameOrFirstname(String
           lastname, String firstname);
8
9       // Enabling ignoring case for an individual property
10      List<Person> findByLastnameIgnoreCase(String
           lastname);
11      // Enabling ignoring case for all suitable properties
```

```
12        List<Person>
             findByLastnameAndFirstnameAllIgnoreCase(String
             lastname, String firstname);
13
14        // Enabling static ORDER BY for a query
15        List<Person>
             findByLastnameOrderByFirstnameAsc(String
             lastname);
16        List<Person>
             findByLastnameOrderByFirstnameDesc(String
             lastname);
17    }
```

13.2 Reserved Method Names

Reserved methods like CrudRepository.findById (or just findById) are targeting the identifier property regardless of the actual property name used in the declared method. Example:

```
1     class User {
2         //The identifier property (primary key).
3         @Id Long pk;
4
5         // A property named id, but not the identifier.
6         Long id;
7     }
8
9     interface UserRepository extends Repository<User, Long>
         {
10
11        // Targets the pk property (the one marked with @Id
             which is considered to be the identifier) as it
             refers to a CrudRepository base repository
             method.
12        Optional<User> findById(Long id);
13
14        // Targets the pk property by name as it is a
             derived query.
15        Optional<User> findByPk(Long pk);
16
17        // Targets the id property by using the descriptive
             token between find and by to avoid collisions
             with reserved methods.
```

```
18          Optional<User> findUserById(Long id);
19      }
```

13.3 Using explicitly defined queries

To explicitly define queries, the annotation @Query can be used.

The queries themselves are tied to the Java method that executes them, you can actually bind them directly by using the Spring Data JPA @Query annotation rather than annotating them to the domain class. This frees the domain class from persistence specific information and co-locates the query to the repository interface.

Queries annotated to the query method take precedence over queries defined using @NamedQuery or named queries declared in orm.xml.

Example:

```
1     public interface UserRepository extends
          JpaRepository<User, Long> {
2
3     @Query("select u from User u where u.emailAddress = ?1")
4     User findByEmailAddress(String emailAddress);
5 }
```

13.4 Paging, Iterating Large Results, Sorting and Limiting

Spring recognizes certain specific types like Pageable, Sort and Limit, to apply pagination, sorting and limiting to your queries dynamically. Example:

```
1     Page<User> findByLastname(String lastname, Pageable
          pageable);
2
3     Slice<User> findByLastname(String lastname, Pageable
          pageable);
4
5     List<User> findByLastname(String lastname, Sort sort);
6
7     List<User> findByLastname(String lastname, Sort sort,
          Limit limit);
8
9     List<User> findByLastname(String lastname, Pageable
          pageable);
```

13.5 Repository Query Keywords

```
1     // General query method returning typically the
         repository type, a Collection or Streamable subtype
         or a result wrapper such as Page, GeoResults or any
         other store-specific result wrapper. Can be used as
         findBy..., findMyDomainTypeBy... or in combination
         with additional keywords.
2     find...By, read...By, get...By, query...By,
         search...By, stream...By
3
4     // Exists projection, returning typically a boolean
         result.
5     exists...By
6
7     // Count projection returning a numeric result.
8     count...By
9
10    //  Delete query method returning either no result
         (void) or the delete count.
11    delete...By, remove...By
12
13    //  Limit the query results to the first <number> of
         results. This keyword can occur in any place of the
         subject between find (and the other keywords) and by.
14    ...First<number>..., ...Top<number>...
15
16    //  Use a distinct query to return only unique results.
         Consult the store-specific documentation whether
         that feature is supported. This keyword can occur in
         any place of the subject between find (and the other
         keywords) and by.
17    ...Distinct...
```

13.6 Supported query method predicate keywords and modifiers

In addition to filter predicates, the following list of modifiers is supported:

- IgnoreCase, IgnoringCase
- AllIgnoreCase, AllIgnoringCase
- OrderBy... (e. g. OrderByFirstnameAscLastnameDesc).

Logical keyword	Keyword expressions
AND	And
OR	Or
AFTER	After, IsAfter
BEFORE	Before, IsBefore
CONTAINING	Containing, IsContaining, Contains
BETWEEN	Between, IsBetween
ENDING_WITH	EndingWith, IsEndingWith, EndsWith
EXISTS	Exists
FALSE	False, IsFalse
GREATER_THAN	GreaterThan, IsGreaterThan
GREATER_THAN_EQUALS	GreaterThanEqual, IsGreaterThanEqual
IN	In, IsIn
IS	Is, Equals, (or no keyword)
IS_EMPTY	IsEmpty, Empty
IS_NOT_EMPTY	IsNotEmpty, NotEmpty
IS_NOT_NULL	NotNull, IsNotNull
IS_NULL	Null, IsNull
LESS_THAN	LessThan, IsLessThan
LESS_THAN_EQUAL	LessThanEqual, IsLessThanEqual
LIKE	Like, IsLike
NEAR	Near, IsNear
NOT	Not, IsNotNOT_INNotIn, IsNotIn
NOT_LIKE	NotLike, IsNotLike
REGEX	Regex, MatchesRegex, Matches
STARTING_WITH	StartingWith, IsStartingWith, StartsWith
TRUE	True, IsTrue
WITHIN	Within, IsWithin

14 Spring Security

14.1 Overview

Architecture Overview (see 12):

14.2 Filter Chain

Overview (see 13):
  Example (see 14):

14.3 Authentication

Overview (see 15):

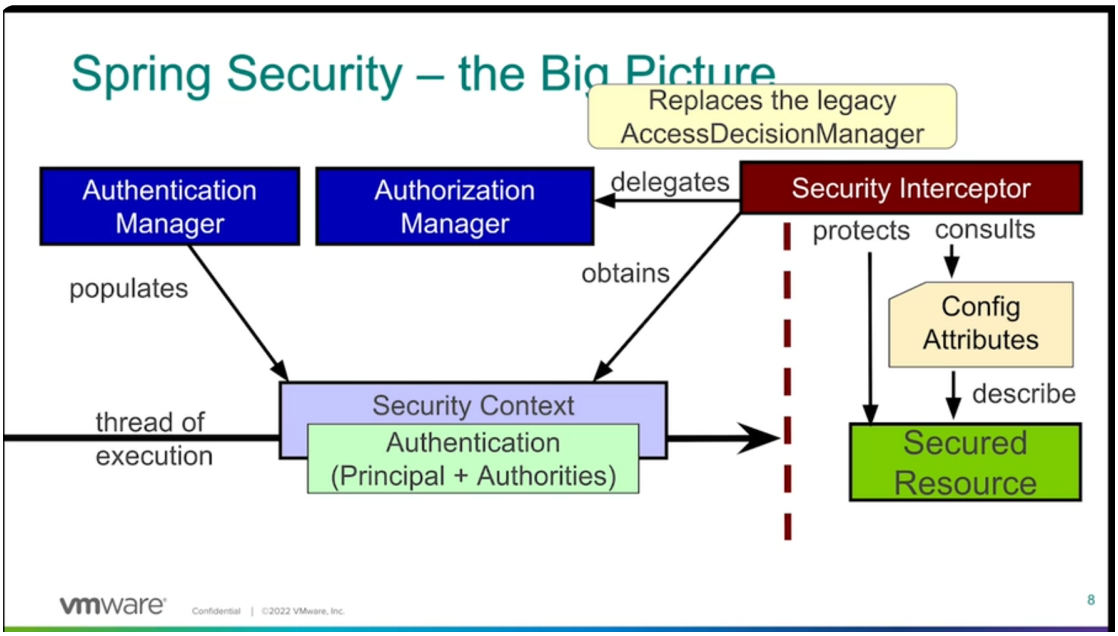


Figure 12: Spring Security Architecture Overview

14.4 Authorization

All Authentication implementations store a list of GrantedAuthority objects. These represent the authorities that have been granted to the principal. The GrantedAuthority objects are inserted into the Authentication object by the AuthenticationManager and are later read by AuthorizationManager instances when making authorization decisions.

The GrantedAuthority interface has only one method:

```
1     String getAuthority();
```

String getAuthority();

This method is used by an AuthorizationManager instance to obtain a precise String representation of the GrantedAuthority.

Spring Security includes one concrete GrantedAuthority implementation: SimpleGrantedAuthority. All AuthenticationProvider instances included with the security architecture use SimpleGrantedAuthority to populate the Authentication object.

By default, role-based authorization rules include ROLE_ as a prefix. You can customize this with GrantedAuthorityDefaults.

You can configure the authorization rules to use a different prefix by exposing a GrantedAuthorityDefaults bean, like so:

```
1     @Bean
2     static GrantedAuthorityDefaults
         grantedAuthorityDefaults() {
```

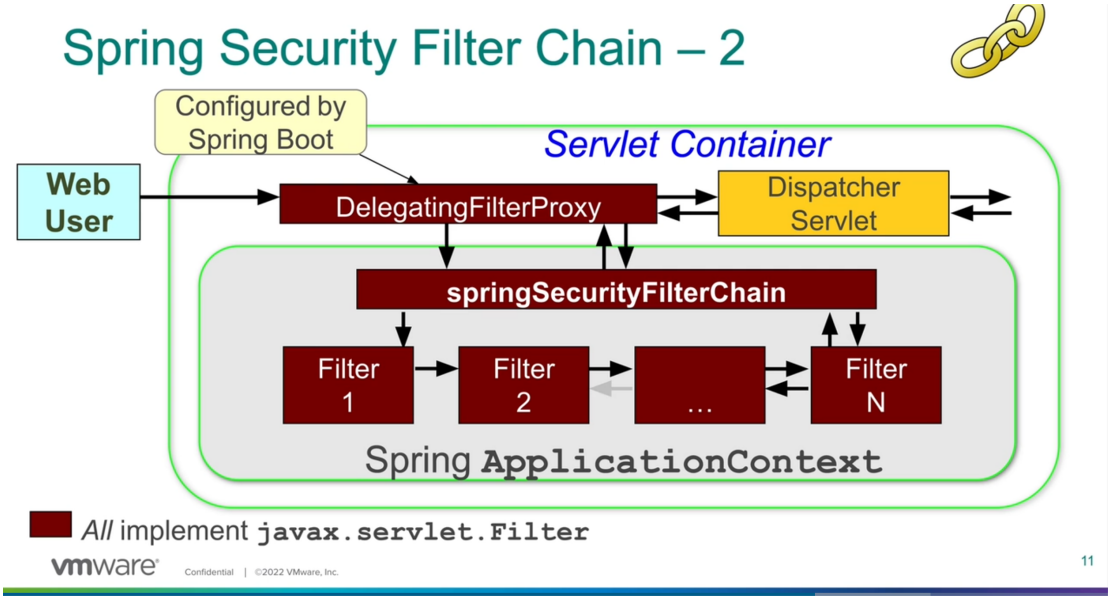


Figure 13: Spring Security Filter Chain

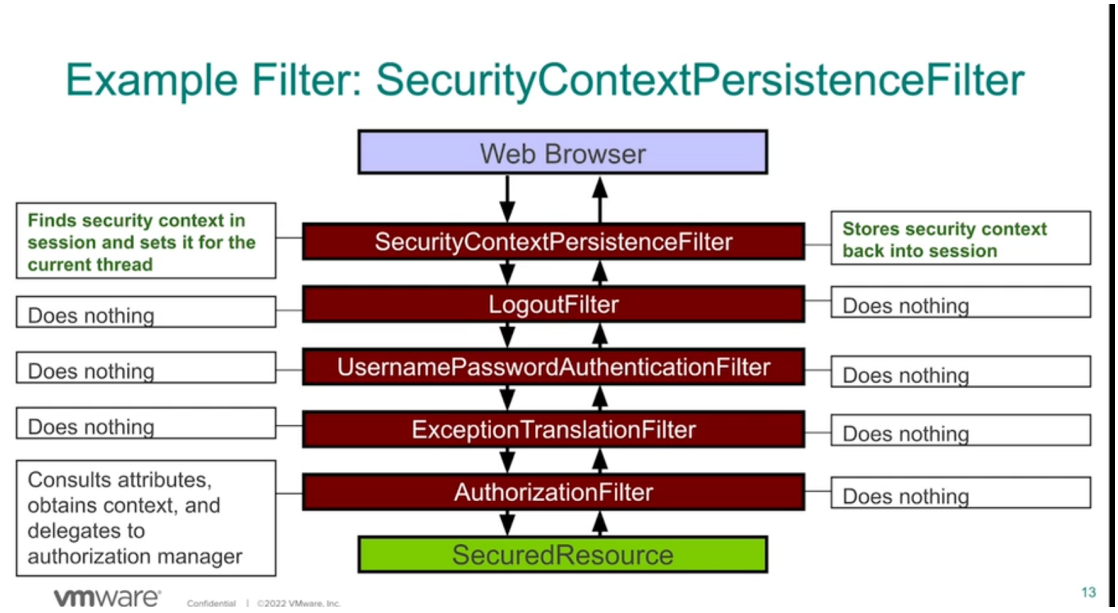


Figure 14: Example Filter Chain

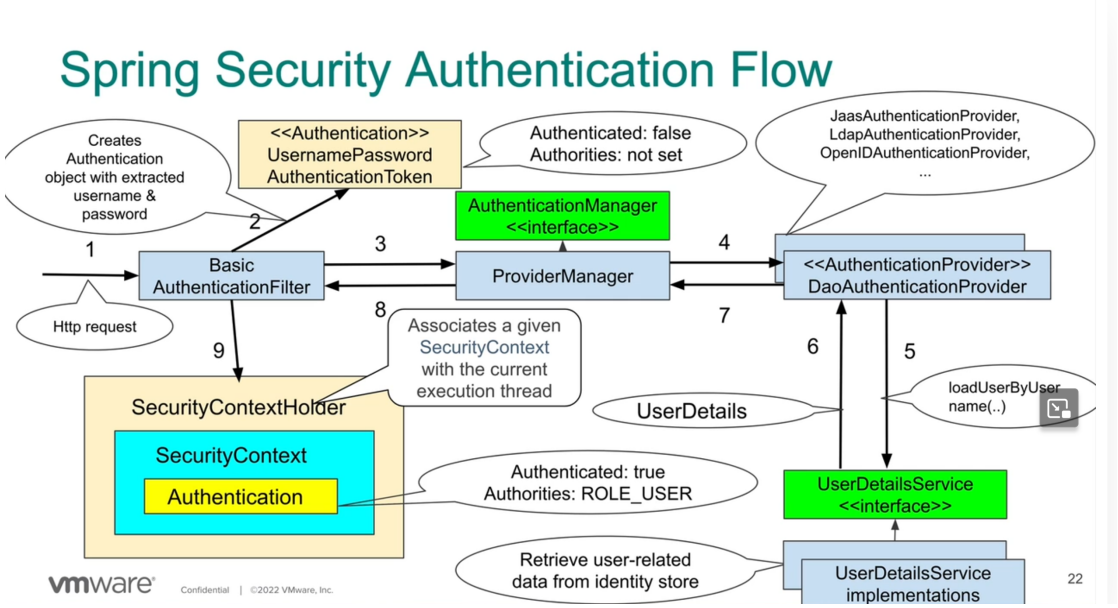


Figure 15: Spring Security Filter Chain

```
3         return new GrantedAuthorityDefaults("MYPREFIX_");
4     }
```

You expose GrantedAuthorityDefaults using a static method to ensure that Spring publishes it before it initializes Spring Security's method security @Configuration classes.

14.4.1 Invocation Handling

Spring Security provides interceptors that control access to secure objects, such as method invocations or web requests. A pre-invocation decision on whether the invocation is allowed to proceed is made by AuthorizationManager instances. Also post-invocation decisions on whether a given value may be returned is made by AuthorizationManager instances.

AuthorizationManagers are called by Spring Security's request-based, method-based, and message-based authorization components and are responsible for making final access control decisions.

The AuthorizationManager interface contains two methods:

```
1     AuthorizationDecision check(Supplier<Authentication>
          authentication, Object secureObject);
2
3     default void verify(Supplier<Authentication>
          authentication, Object secureObject)
4     throws AccessDeniedException {
```

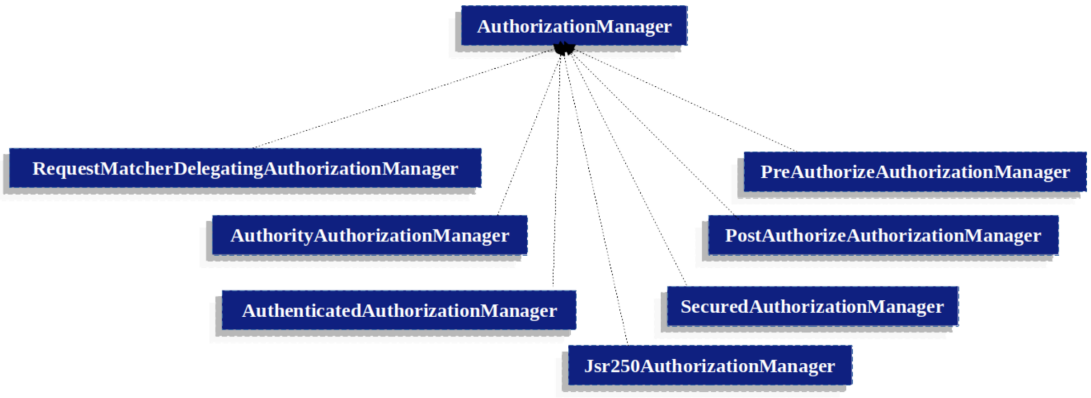


Figure 16: Spring Security Filter Chain

```
5           // ...
6       }
```

The AuthorizationManager's check method is passed all the relevant information it needs in order to make an authorization decision. In particular, passing the secure Object enables those arguments contained in the actual secure object invocation to be inspected. For example, let's assume the secure object was a MethodInvocation. It would be easy to query the MethodInvocation for any Customer argument, and then implement some sort of security logic in the AuthorizationManager to ensure the principal is permitted to operate on that customer. Implementations are expected to return a positive AuthorizationDecision if access is granted, negative AuthorizationDecision if access is denied, and a null AuthorizationDecision when abstaining from making a decision.

verify calls check and subsequently throws an AccessDeniedException in the case of a negative AuthorizationDecision.

Here is an overview of AuthorizationManager implementations: see 16.

14.4.2 Method Security

Spring Security also supports modeling at the method level.

You can activate it in your application by annotating any @Configuration class with @EnableMethodSecurity or adding <method-security >to any XML configuration file. (Note: Spring Boot Starter Security does not activate method-level authorization by default.)

Then, you are immediately able to annotate any Spring-managed class or method with @PreAuthorize, @PostAuthorize, @PreFilter, and @PostFilter to authorize method invocations, including the input parameters and return values.

Spring Security's method authorization support is handy for:

- Extracting fine-grained authorization logic; for example, when the method parameters and return values contribute to the authorization decision.
- Enforcing security at the service layer.
- Stylistically favoring annotation-based over HttpSecurity-based configuration.

And since Method Security is built using Spring AOP, you have access to all its expressive power to override Spring Security's defaults as needed.

Example:

```
1  @Service
2  public class MyCustomerService {
3      @PreAuthorize("hasAuthority('permission:read')")
4      @PostAuthorize("returnObject.owner ==
           authentication.name")
5      public Customer readCustomer(String id) { ... }
6  }
```

15 Testing with Spring

15.1 TestContext Framework

The Spring TestContext Framework (located in the org.springframework.test.context package) provides generic, annotation-driven unit and integration testing support that is agnostic of the testing framework in use.

In addition to generic testing infrastructure, the TestContext framework provides explicit support for JUnit 4, JUnit Jupiter (AKA JUnit 5), and TestNG.

15.2 Executing SQL Scripts

Sometimes it is essential to be able to modify the database during integration tests. Spring provides the following options for executing SQL scripts programmatically within integration test methods:

- org.springframework.jdbc.datasource.init.ScriptUtils
- org.springframework.jdbc.datasource.init.ResourceDatabasePopulator
- org.springframework.test.context.junit4.AbstractTransactionalJUnit4SpringContextTests
- org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTests

`ScriptUtils` provides a collection of static utility methods (all called `executeSqlScript`) for working with SQL scripts and is mainly intended for internal use within the framework. However, if you require full control over how SQL scripts are parsed and run, ScriptUtils may suit your needs better than some of the other alternatives described later.

```
1     // one of 4 overloaded versions
2     public static void executeSqlScript(
3         Connection connection,
4         EncodedResource resource,
5         boolean continueOnError,
6         boolean ignoreFailedDrops,
7         String commentPrefix,
8         @Nullable
9         String separator,
10        String blockCommentStartDelimiter,
11        String blockCommentEndDelimiter
12    ) throws ScriptException
```

`ResourceDatabasePopulator` provides an object-based API for programmatically populating, initializing, or cleaning up a database by using SQL scripts defined in external resources. ResourceDatabasePopulator provides options for configuring the character encoding, statement separator, comment delimiters, and error handling flags used when parsing and running the scripts.

```
1     //  Populates, initializes, or cleans up a database
         using SQL scripts defined in external resources.
2
3     public class ResourceDatabasePopulator implements
         DatabasePopulator
4
5     /*
6     Call addScript(org.springframework.core.io.Resource) to
         add a single SQL script location.
7     Call
         addScripts(org.springframework.core.io.Resource...)
         to add multiple SQL script locations.
8     Consult the setter methods in this class for further
         configuration options.
9     Call populate(java.sql.Connection) or
         execute(javax.sql.DataSource) to initialize or clean
         up the database using the configured scripts.
10    */
```

In addition to the aforementioned mechanisms for running SQL scripts programmatically, you can declare the `@Sql` annotation on a test class or test method to configure individual SQL statements or the resource paths to SQL scripts that should be run against a given database before or after an integration test class or test method. Support for @Sql is provided by the `SqlScriptsTestExecutionListener`, which is enabled by default.

15.3 MockMvc

The Spring MVC Test framework, also known as MockMvc, aims to provide more complete testing for Spring MVC controllers without a running server. It does that by invoking the DispatcherServlet and passing "mock" implementations of the Servlet API from the spring-test module which replicates the full Spring MVC request handling without a running server.

15.3.1 Implications

As opposed to @SpringBootTest, MockMvc is built on Servlet API mock implementations from the spring-test module and does not rely on a running container.

MockMvc starts out with a blank MockHttpServletRequest. Whatever is added to it is what the request becomes. There is no jsessionid cookie; no forwarding, error, or async dispatches; and no JSP rendering. Instead, "forwarded" and "redirected" URLs are saved in the MockHttpServletResponse and can be *asserted with expectations*.

This means that, if you use JSPs, you can verify the JSP page to which the request was forwarded, but no HTML is rendered. In other words, the JSP is not invoked. Note, however, that all other rendering technologies that do not rely on forwarding, such as Thymeleaf and Freemarker, render HTML to the response body as expected. The same is true for rendering JSON, XML, and other formats through @ResponseBody methods.

15.3.2 Static imports

When using MockMvc directly to perform requests, the following static imports are needed:

- MockMvcBuilders.*
- MockMvcRequestBuilders.*
- MockMvcResultMatchers.*
- MockMvcResultHandlers.*

15.3.3 Setup

MockMvc can be setup in one of two ways. One is to point directly to the controllers you want to test and programmatically configure Spring MVC infrastructure. Example:

```
1     class MyWebTests {
2
3         MockMvc mockMvc;
4
5         @BeforeEach
6         void setup() {
7             this.mockMvc =
                  MockMvcBuilders.standaloneSetup(new
                  AccountController()).build();
8         }
9
10        // ...
11
12    }
```

The second is to point to Spring configuration with Spring MVC and controller infrastructure in it.

```
1     @SpringJUnitWebConfig(locations =
         "my-servlet-context.xml")
2     class MyWebTests {
3
4         MockMvc mockMvc;
5
6         @BeforeEach
7         void setup(WebApplicationContext wac) {
8             this.mockMvc =
                  MockMvcBuilders.webAppContextSetup(wac).build();
9         }
10
11        // ...
12
13    }
```

15.3.4 Queries with MockMvc

Example queries using MockMvc:

```
1     mockMvc.perform(post("/hotels/{id}",
         42).accept(MediaType.APPLICATION_JSON));
2
3     // a file upload request that internally uses
         MockMultipartHttpServletRequest
```

```
4     mockMvc.perform(multipart("/doc").file("a1",
          "ABC".getBytes("UTF-8")));
5
6     // specifying query parameters in URI template style
7     mockMvc.perform(get("/hotels?thing={thing}",
          "somewhere"));
8
9     // adding Servlet request parameters that represent
          either query or form parameters
10    mockMvc.perform(get("/hotels").param("thing",
          "somewhere"));
```

15.4 Mocking in detail: @Mock vs. @MockBean

@Mock is an annotation provided by the Mockito library. It is used to create mock objects for dependencies that are not part of the Spring context.

The @Mock annotation is typically used in conjunction with the MockitoJUnitRunner or MockitoExtension to initialize the mock objects.

Example:

```
1     import static org.mockito.Mockito.*;
2
3     @RunWith(MockitoJUnitRunner.class)
4     public class UserServiceTest {
5
6         @Mock
7         private UserRepository userRepository;
8
9         // inject mock objects into UserService
10        @InjectMocks
11        private UserService userService;
12
13        @Test
14        public void testGetUserById() {
15            // Given
16            User mockedUser = new User("John", "Doe", 25);
17            when(userRepository.findById(1L)).thenReturn(
18            Optional.of(mockedUser));
19
20            // When
21            User result = userService.getUserById(1L);
22
23            // Then
```

```
24              assertNotNull(result);
25              assertEquals("John", result.getFirstName());
26
27              // Verify that the findById method was called
28              verify(userRepository).findById(1L);
29          }
30      }
```

In contrast, @MockBean is a Spring Boot-specific annotation provided by the Spring Boot Test module. It is used to create mock objects for dependencies that are part of the Spring context. Example:

```
1       @SpringBootTest
2       public class UserServiceIntegrationTest {
3
4           @Autowired
5           private UserService userService;
6
7           @MockBean
8           private UserRepository userRepository;
9
10          @Test
11          public void testGetUserById() {
12              // same as above
13          }
14      }
```

Key differences:

- @Mock can only be applied to fields and parameters, whereas @MockBean can only be applied to classes and fields.
- @Mock can be used to mock any Java class or interface while @MockBean only allows for mocking of Spring beans or creation of mock Spring beans. It can be used to mock existing beans, but also to create new beans that will belong to the Spring application context.
- To be able to use the @MockBean annotation, the Spring runner (@RunWith(SpringRunner.class)) is used, whereas @Mock is used with MockitoJUnitRunner.
- @MockBean can be used to create custom annotations for specific, reoccurring, needs.

Both @Mock and @MockBean are included in spring-boot-starter-test.

16 Spring Web MVC

Spring Web MVC is the original web framework built on the Servlet API and has been included in the Spring Framework from the very beginning.

16.1 Controllers

@RequestMapping handler methods have a flexible signature and can choose from a range of supported controller method arguments and return values.

Supported controller method arguments are:

```
1     // Generic access to request parameters and request and
          session attributes, without direct use of the
          Servlet API.
2     WebRequest
3
4     // Choose any specific request or response type -for
          example, ServletRequest, HttpServletRequest, or
          Spring's MultipartRequest,
          MultipartHttpServletRequest.
5     jakarta.servlet.ServletRequest,
          jakarta.servlet.ServletResponse
6
7     jakarta.servlet.http.HttpSession
8
9     java.security.Principal
10
11    HttpMethod
12
13    // Java
14    java.util.Locale
15
16    java.util.TimeZone + java.time.ZoneId
17
18
19
20    // For access to the raw request body as exposed by the
          Servlet API.
21    java.io.InputStream, java.io.Reader
22
23    // For access to the raw response body as exposed by
          the Servlet API.
24    java.io.OutputStream, java.io.Writer
25
26    @PathVariable
```

```
27      //For access to name-value pairs in URI path segments.
28      @MatrixVariable
29
30      @RequestParam
31      // For access to the Servlet request parameters,
           including multipart files. Parameter values are
           converted to the declared method argument type.
32
33      // For access to request headers. Header values are
           converted to the declared method argument type,
34      @RequestHeader
35
36      @CookieValue
37
38      // For access to the HTTP request body. Body content is
           converted to the declared method argument type by
           using HttpMessageConverter implementations.
39      @RequestBody
40
41      // For access to request headers and body. The body is
           converted with an HttpMessageConverter.
42      HttpEntity<B>
43
44      // For access to a part in a multipart/form-data
           request, converting the part's body with an
           HttpMessageConverter.
45      @RequestPart
46
47      // For access to the model that is used in HTML
           controllers and exposed to templates as part of view
           rendering.
48      java.util.Map, org.springframework.ui.Model,
           org.springframework.ui.ModelMap
49
50      // Attributes to use in case of a redirect (that is, to
           be appended to the query string) and flash
           attributes to be stored temporarily until the
           request after redirect.
51      RedirectAttributes
52
53      // For access to an existing attribute in the model
           (instantiated if not present) with data binding and
           validation applied. See @ModelAttribute as well as
           Model and DataBinder.
```

```
54      @ModelAttribute
55
56      // For access to errors from validation and data
           binding for a command object (that is, a
           @ModelAttribute argument) or errors from the
           validation of a @RequestBody or @RequestPart
           arguments.
57      Errors, BindingResult
58
59      // For preparing a URL relative to the current
           request's host, port, scheme, context path, and the
           literal part of the servlet mapping. See URI Links.
60      UriComponentsBuilder
61
62      // For access to any session attribute, in contrast to
           model attributes stored in the session as a result
           of a class-level @SessionAttributes declaration.
63      @SessionAttribute
64
65      @RequestAttribute
66
67      // If a method argument is not matched to any of the
           earlier values in this table and it is a simple type
           (as determined by BeanUtils#isSimpleProperty), it is
           resolved as a @RequestParam. Otherwise, it is
           resolved as a @ModelAttribute.
```

Supported return values *include*:

```
1       // The return value is converted through
           HttpMessageConverter implementations and written to
           the response.
2       @ResponseBody
3
4       // The return value that specifies the full response
           (including HTTP headers and body) is to be converted
           through HttpMessageConverter implementations and
           written to the response.
5       HttpEntity<B>, ResponseEntity<B>
6
7       HttpHeaders
8
9       ErrorResponse
10
```

```
11      // A view name to be resolved with ViewResolver
            implementations and used together with the implicit
            model
12      String
13
14      //  A View instance to use for rendering together with
            the implicit model
15      View
16
17      // The view and model attributes to use and,
            optionally, a response status.
18      ModelAndView
19
20      // Attributes to be added to the implicit model, with
            the view name implicitly determined through a
            RequestToViewNameTranslator.
21
22      java.util.Map, org.springframework.ui.Model
```

16.2 Type Conversion

By default, formatters for various number and date types are installed, along with support for customization via @NumberFormat, @DurationFormat, and @DateTimeFormat on fields and parameters.

To register custom formatters and converters, use the following:

```
1       @Configuration
2       public class WebConfiguration implements
            WebMvcConfigurer {
3
4           @Override
5           public void addFormatters(FormatterRegistry
                registry) {
6               // ...
7           }
8       }
```

16.3 Validation

JSR-303/JSR-380 annotations provide a standard way of validating objects in Java. These annotations can be used on domain object fields to enforce validation constraints.

Example:

```
1     public class User {
2
3         @NotNull
4         @Size(min = 1, max = 20)
5         private String name;
6     }
```

In addition, if Bean Validation is present on the classpath (for example, Hibernate Validator), the LocalValidatorFactoryBean is registered as a global Validator for use with @Valid and @Validated on controller method arguments.

You can customize the global Validator instance, as the following example shows:

```
1     @Configuration
2     public class WebConfiguration implements
         WebMvcConfigurer {
3
4         @Override
5         public Validator getValidator() {
6             Validator validator = new
                 OptionalValidatorFactoryBean();
7             // ...
8             return validator;
9         }
10    }
```

Note that you can also register Validator implementations locally, as the following example shows:

```
1     @Controller
2     public class MyController {
3
4         @InitBinder
5         public void initBinder(WebDataBinder binder) {
6             binder.addValidators(new FooValidator());
7         }
8     }
```

16.4 Interceptors

You can register interceptors to apply to incoming requests, as the following example shows:

```
1     @Configuration
2     public class WebConfiguration implements
         WebMvcConfigurer {
3
4         @Override
5         public void addInterceptors(InterceptorRegistry
             registry) {
6             registry.addInterceptor(new
                 LocaleChangeInterceptor());
7         }
8     }
```

16.5 Content Types

You can configure how Spring MVC determines the requested media types from the request (for example, Accept header, URL path extension, query parameter, and others).

By default, only the Accept header is checked.

You can customize requested content type resolution, as the following example shows:

```
1     @Configuration
2     public class WebConfiguration implements
         WebMvcConfigurer {
3
4         @Override
5         public void configureContentNegotiation(
6         ContentNegotiationConfigurer configurer) {
7             configurer.mediaType("json",
                 MediaType.APPLICATION_JSON);
8             configurer.mediaType("xml",
                 MediaType.APPLICATION_XML);
9         }
10    }
```

16.6 Message Converters

You can set the HttpMessageConverter instances to use in Java configuration, replacing the ones used by default, by overriding configureMessageConverters(). You can also

customize the list of configured message converters at the end by overriding extendMessageConverters().

In a Spring Boot application, the WebMvcAutoConfiguration adds any HttpMessageConverter beans it detects, in addition to default converters. Hence, in a Boot application, prefer to use the HttpMessageConverters mechanism.

Or alternatively, use extendMessageConverters to modify message converters at the end.

The following example adds XML and Jackson JSON converters with a customized ObjectMapper instead of the default ones:

```
1    @Configuration
2    public class WebConfiguration implements
        WebMvcConfigurer {
3
4        @Override
5        public void configureMessageConverters(
6        List<HttpMessageConverter<?>> converters) {
7            Jackson2ObjectMapperBuilder builder = new
                Jackson2ObjectMapperBuilder()
8            .indentOutput(true)
9            .dateFormat(new SimpleDateFormat("yyyy-MM-dd"))
10           .modulesToInstall(new ParameterNamesModule());
11           converters.add(new
                MappingJackson2HttpMessageConverter(
12           builder.build()));
13           converters.add(new
                MappingJackson2XmlHttpMessageConverter(
14           builder.createXmlMapper(true).build()));
15       }
16   }
```

16.7 HTTP Message Conversion

The spring-web module contains the HttpMessageConverter interface for reading and writing the body of HTTP requests and responses through InputStream and OutputStream. HttpMessageConverter instances are used on the client side (for example, in the RestClient) and on the server side (for example, in Spring MVC REST controllers).

Concrete implementations for the main media (MIME) types are provided in the framework and are, by default, registered with the RestClient and RestTemplate on the client side and with RequestMappingHandlerAdapter on the server side (see Configuring Message Converters).

Some implementations of HttpMessageConverter are:

```
    StringHttpMessageConverter
    FormHttpMessageConverter
    ByteArrayHttpMessageConverter
    MarshallingHttpMessageConverter
    JsonbHttpMessageConverter
    ProtobufHttpMessageConverter
    ProtobufJsonFormatHttpMessageConverter
```

16.8 URI Links

16.8.1 UriComponents

UriComponentsBuilder helps to build URI's from URI templates with variables.

```
    UriComponents uriComponents = UriComponentsBuilder
    .fromUriString("https://example.com/hotels/{hotel}")
    .queryParam("q", "{q}")
    .encode()
    .build();

    URI uri = uriComponents.expand("Westin", "123").toUri();
```

The preceding example can be consolidated into one chain and shortened with buildAndExpand:

```
    URI uri = UriComponentsBuilder
    .fromUriString("https://example.com/hotels/{hotel}")
    .queryParam("q", "{q}")
    .encode()
    .buildAndExpand("Westin", "123")
    .toUri();
```

You can shorten it further by going directly to a URI (which implies encoding):

```
    URI uri = UriComponentsBuilder
    .fromUriString("https://example.com/hotels/{hotel}")
    .queryParam("q", "{q}")
    .build("Westin", "123");
```

You can shorten it further still with a full URI template:

```
    URI uri = UriComponentsBuilder
    .fromUriString("https://example.com/hotels/{hotel}?q={q}")
    .build("Westin", "123");
```

16.8.2 UriBuilder

UriComponentsBuilder implements UriBuilder. You can create a UriBuilder, in turn, with a UriBuilderFactory. Together, UriBuilderFactory and UriBuilder provide a pluggable mechanism to build URIs from URI templates, based on shared configuration, such as a base URL, encoding preferences, and other details.

You can configure RestTemplate and WebClient with a UriBuilderFactory to customize the preparation of URIs. DefaultUriBuilderFactory is a default implementation of UriBuilderFactory that uses UriComponentsBuilder internally and exposes shared configuration options.

The following example shows how to configure a RestTemplate:

```
    // import org.springframework.web.util.
        DefaultUriBuilderFactory.EncodingMode;

    String baseUrl = "https://example.org";
    DefaultUriBuilderFactory factory = new
       DefaultUriBuilderFactory(baseUrl);
    factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

    RestTemplate restTemplate = new RestTemplate();
    restTemplate.setUriTemplateHandler(factory);

    In addition, you can also use DefaultUriBuilderFactory
       directly. It is similar to using
       UriComponentsBuilder but, instead of static factory
       methods, it is an actual instance that holds
       configuration and preferences.

    String baseUrl = "https://example.com";
    DefaultUriBuilderFactory uriBuilderFactory = new
       DefaultUriBuilderFactory(baseUrl);

    URI uri = uriBuilderFactory.uriString("/hotels/{hotel}")
    .queryParam("q", "{q}")
    .build("Westin", "123");
```

16.9 View Controllers

This is a shortcut for defining a ParameterizableViewController that immediately forwards to a view when invoked. You can use it in static cases when there is no Java controller logic to run before the view generates the response.

The following example forwards a request for / to a view called home:

```
1    @Configuration
2    public class WebConfiguration implements
        WebMvcConfigurer {
3
4        @Override
5        public void
            addViewControllers(ViewControllerRegistry
            registry) {
6            registry
7            .addViewController("/")
8            .setViewName("home");
9        }
10   }
```

16.10 View Resolvers

The MVC configuration simplifies the registration of view resolvers.

The following example configures content negotiation view resolution by using JSP and Jackson as a default View for JSON rendering:

```
1    @Configuration
2    public class WebConfiguration implements
        WebMvcConfigurer {
3
4        @Override
5        public void
            configureViewResolvers(ViewResolverRegistry
            registry) {
6            registry.enableContentNegotiation(new
                MappingJackson2JsonView());
7            registry.jsp();
8        }
9    }
```

16.11 Template Engines

As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies, including Thymeleaf, FreeMarker, and JSPs. Also, many other templating engines include their own Spring MVC integrations.

Spring Boot includes auto-configuration support for the following templating engines:

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

If possible, JSPs should be avoided. There are several known limitations when using them with embedded servlet containers.

When you use one of these templating engines with the default configuration, your templates are picked up automatically from src/main/resources/templates.

16.12 MVC Config

The MVC Java configuration and the MVC XML namespace provide default configuration suitable for most applications and a configuration API to customize it.

You can use the @EnableWebMvc annotation to enable MVC configuration with programmatic configuration, or <mvc:annotation-driven >with XML configuration:

```
1     @Configuration
2     @EnableWebMvc
3     public class WebConfiguration {
4     }
```

The preceding example registers a number of Spring MVC infrastructure beans and adapts to dependencies available on the classpath (for example, payload converters for JSON, XML, and others).

16.13 MVC Config API

In Java configuration, you can implement the WebMvcConfigurer interface:

```
1     public class WebConfiguration implements
         WebMvcConfigurer {
2
3          // Implement configuration methods...
4     }
```

@

17 REST Clients

- RestClient is a synchronous HTTP client that exposes a modern, fluent API.
- WebClient is a reactive client to perform HTTP requests with a fluent API.
- RestTemplate is a synchronous client to perform HTTP requests. It is the original Spring REST client and exposes a simple, template-method API over underlying HTTP client libraries.
- HTTP Interface: The Spring Frameworks lets you define an HTTP service as a Java interface with HTTP exchange methods. You can then generate a proxy that implements this interface and performs the exchanges. This helps to simplify HTTP remote access and provides additional flexibility for to choose an API style such as synchronous or reactive.

Here, we zoom in on RestTemplate.

The RestTemplate provides a high-level API over HTTP client libraries in the form of a classic Spring Template class. It exposes the following groups of overloaded methods:

```
1 // Retrieves a representation via GET.
2 getForObject
3
4 // Retrieves a ResponseEntity (that is, status, headers,
    and body) by using GET.
5 getForEntity
6
7 headForHeaders
8
9 // Creates a new resource by using POST and returns the
    Location header from the response.
10 postForLocation
11
12 // Creates a new resource by using POST and returns the
    representation from the response.
13 postForObject
14
15 // Creates a new resource by using POST and returns the
    representation from the response.
16 postForEntity
17
18 put
19
20 patchForObject
21
22 delete
```

```
23
24 optionsForAllow
25
26 // More generalized (and less opinionated) version of the
      preceding methods that provides extra flexibility when
      needed. It accepts a RequestEntity (including HTTP
      method, URL, headers, and body as input) and returns a
      ResponseEntity.
27 exchange
28
29 // The most generalized way to perform a request, with full
      control over request preparation and response extraction
      through callback interfaces.
30 execute
```

RestTemplate uses the same HTTP library abstraction as RestClient. By default, it uses the SimpleClientHttpRequestFactory, but this can be changed via the constructor.

RestTemplate can be instrumented for observability, in order to produce metrics and traces.

Objects passed into and returned from RestTemplate methods are converted to and from HTTP messages with the help of an HttpMessageConverter.

18 Integration Features

18.1 Task Execution and Scheduling

Spring provides annotation support for both task scheduling and asynchronous method execution.

To enable support for @Scheduled and @Async annotations, you can add @EnableScheduling and @EnableAsync to one of your @Configuration classes, or <task:annotation-driven >element:

```
1     @Configuration
2     @EnableAsync
3     @EnableScheduling
4     public class SchedulingConfiguration {
5     }
```

@Scheduled usage example:

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

```
4     }
```

19 Spring Boot (Web): Servlet Web Applications

If you want to build servlet-based web applications, you can take advantage of Spring Boot's auto-configuration for Spring MVC or Jersey.

The *Spring Web MVC framework* (cf. fig. 18) is a rich "model view controller" web framework. Spring MVC lets you create special @Controller or @RestController beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP by using @RequestMapping annotations.

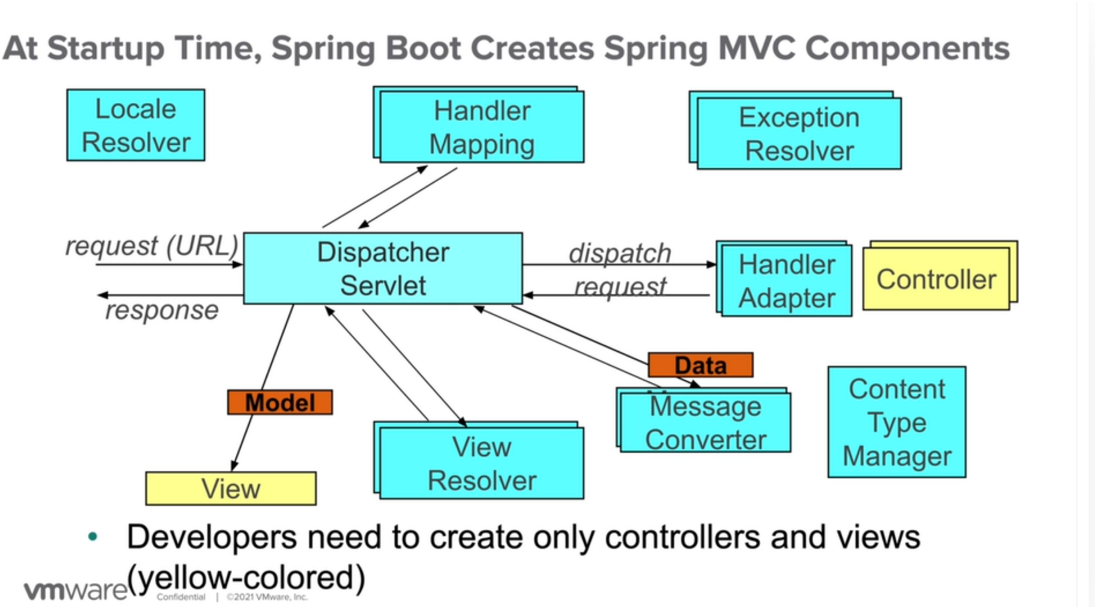


Figure 17: Spring Boot Web Architecture Overview

Spring Boot provides auto-configuration for Spring MVC that works well with most applications. It replaces the need for @EnableWebMvc, and the two cannot be used together. In addition to Spring MVC's defaults, the auto-configuration provides the following features:

- `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
- Support for serving static resources, including support for WebJars (covered later in this document).
- Automatic registration of Converter, GenericConverter, and Formatter beans.
- Support for HttpMessageConverters (covered later in this document).

- Automatic registration of `MessageCodesResolver`.
- Static index.html support.
- Automatic use of a `ConfigurableWebBindingInitializer` bean

19.1 Spring Boot Starters

Main starter:

```
// spring-boot-starter/build.gradle

plugins {
    id "org.springframework.boot.starter"
}

description = "Core starter, including
   auto-configuration support, logging and YAML"

dependencies {
    api(project(":spring-boot-project:spring-boot"))
    api(project(":spring-boot-project:
    spring-boot-autoconfigure"))
    api(project(":spring-boot-project:
    spring-boot-starters:spring-boot-starter-logging"))
    api("jakarta.annotation:jakarta.annotation-api")
    api("org.springframework:spring-core")
    api("org.yaml:snakeyaml")
}
```

Examples:

```
// spring-boot-starter-test/build.gradle

plugins {
    id "org.springframework.boot.starter"
}

description = "Starter for testing Spring Boot
   applications with libraries including JUnit Jupiter,
   Hamcrest and Mockito"

dependencies {
    api(project(":spring-boot-project:spring-boot-starters:
    spring-boot-starter"))
```

```
12          api(project(":spring-boot-project:spring-boot-test"))
13          api(project(":spring-boot-project:
14          spring-boot-test-autoconfigure"))
15          api("com.jayway.jsonpath:json-path")
16          api("jakarta.xml.bind:jakarta.xml.bind-api")
17          api("net.minidev:json-smart")
18          api("org.assertj:assertj-core")
19          api("org.awaitility:awaitility")
20          api("org.hamcrest:hamcrest")
21          api("org.junit.jupiter:junit-jupiter")
22          api("org.mockito:mockito-core")
23          api("org.mockito:mockito-junit-jupiter")
24          api("org.skyscreamer:jsonassert")
25          api("org.springframework:spring-core")
26          api("org.springframework:spring-test")
27          api("org.xmlunit:xmlunit-core") {
28              exclude group: "javax.xml.bind", module:
                    "jaxb-api"
29          }
30      }
31
32      checkRuntimeClasspathForConflicts {
33          ignore { name ->
                name.startsWith("mockito-extensions/") }
34      }
```

```
1       // spring-boot-starter-data-jpa/build.gradle
2
3       plugins {
4           id "org.springframework.boot.starter"
5       }
6
7       description = "Starter for using Spring Data JPA with
            Hibernate"
8
9       dependencies {
10          api(project(":spring-boot-project:spring-boot-starters:
11          spring-boot-starter"))
12          api(project(":spring-boot-project:spring-boot-starters:
13
14          spring-boot-starter-jdbc"))
15          api("org.hibernate.orm:hibernate-core")
16          api("org.springframework.data:spring-data-jpa")
```

```
        api("org.springframework:spring-aspects")
    }
```

```
    // spring-boot-starter-aop/build.gradle

    plugins {
        id "org.springframework.boot.starter"
    }

    description = "Starter for aspect-oriented programming
       with Spring AOP and AspectJ"

    dependencies {
        api(project(":spring-boot-project:spring-boot-starters:
        spring-boot-starter"))
        api("org.springframework:spring-aop")
        api("org.aspectj:aspectjweaver")
    }
```

Spring Boot provides integration with three JSON mapping libraries: Gson, Jackson, and JSON-B.

Jackson is the preferred and default library.

19.2 SpringApplication

In addition to the usual Spring Framework events, such as ContextRefreshedEvent, a SpringApplication sends some additional application events.

Some events are actually triggered before the ApplicationContext is created, so you cannot register a listener on those as a @Bean. You can register them with the SpringApplication.addListeners...) method or the SpringApplicationBuilder.listeners(...) method.

If you want those listeners to be registered automatically, regardless of the way the application is created, you can add a `META-INF/spring.factories` file to your project and reference your listener(s) by using the ApplicationListener key, as shown in the following example:

```
    org.springframework.context.ApplicationListener =
       com.example.project.MyListener
```

A Spring Boot application can be shut down programmatically in the following ways:

- use `AbstractApplicationContext.close()`
- call `SpringApplication.exit()`

- register a shutdown hook: `AbstractApplicationContext.registerShutdownHook()`
- use Actuator (need to enable the shutdown endpoint)

19.3 More on spring.factories

The spring.factories file can be used to register custom interface implementations in the following cases:

- Register application event listeners regardless of how the Spring Boot application is created (configured). This is done by implementing a class that inherits from SpringApplicationEvent.
- Locate auto-configuration candidates in, for instance, your own starter module.
- Register a filter to limit the auto-configuration classes considered. See AutoConfigurationImportFilter.
- Activate application listeners that create a file containing the application process id and/or create file(s) containing the port number(s) used by the running web server (if any). These listeners, ApplicationPidFileWriter and WebServerPortFileWriter, both implement the ApplicationListener interface.
- Register failure analyzers. Failure analyzers implement the FailureAnalyzer interface.
- Customize the environment or application context prior to Spring Boot application startup, by implementing the ApplicationListener, ApplicationContextListener or the EnvironmentPostProcessor interfaces.
- Register the availability of view template providers. See TemplateAvailabilityProvider interface.

19.4 ApplicationContext in Boot

A SpringApplication attempts to create the right type of ApplicationContext on your behalf. The algorithm used to determine a WebApplicationType is the following:

- If Spring MVC is present, an AnnotationConfigServletWebServerApplicationContext is used.
- If Spring MVC is not present and Spring WebFlux is present, an AnnotationConfigReactiveWebServerApplicationContext is used.
- Otherwise, AnnotationConfigApplicationContext is used.

It is also possible to take complete control of the ApplicationContext type that is used by calling setApplicationContextFactory(...).

Architecture Overview (see 18):

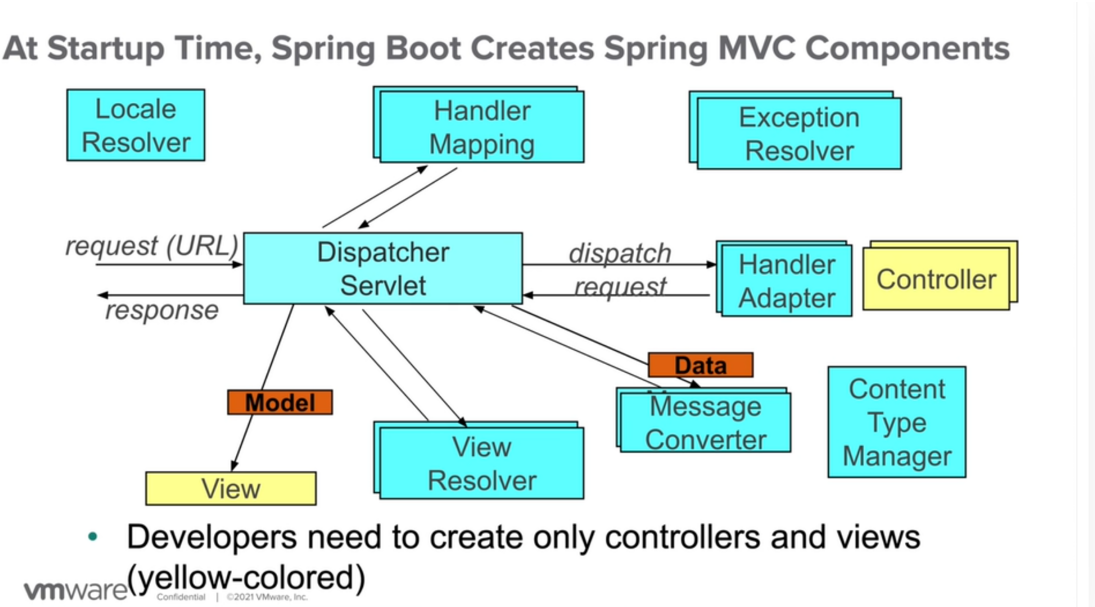


Figure 18: Spring Boot Web Architecture Overview

19.5 Creating Your Own Auto-configuration

19.6 Profiles in Spring Boot

In Spring Boot, the additional annotation @ConfigurationProperties is available. This results in an additional way to use profiles.

In Spring Boot, any @Component, @Configuration or @ConfigurationProperties can be marked with @Profile to limit when it is loaded, as shown in the following example:

```
1  import
     org.springframework.context.annotation.Configuration;
2  import org.springframework.context.annotation.Profile;
3
4  @Configuration(proxyBeanMethods = false)
5  @Profile("production")
6  public class ProductionConfiguration {
7      // ...
8  }
```

Note:

If @ConfigurationProperties beans are registered through @EnableConfigurationProperties instead of automatic scanning, the @Profile annotation needs to be specified on the @Configuration class that has the @EnableConfigurationProperties annotation. In the case where @ConfigurationProperties are scanned, @Profile can be specified on the

@ConfigurationProperties class itself.

19.6.1 Specifying which profiles are active

You can use a spring.profiles.active Environment property to specify which profiles are active. You can specify the property in any of the ways described earlier in this chapter. For example, you could include it in your application.properties, as shown in the following example:

```
spring.profiles.active=dev,hsqldb
```

You could also specify it on the command line by using the following switch: `--spring.profiles.active=`

If no profile is active, a default profile is enabled. The name of the default profile is default and it can be tuned using the `spring.profiles.default` Environment property, as shown in the following example:

```
spring.profiles.default=none
```

spring.profiles.active and spring.profiles.default can only be used in non-profile-specific documents. This means they cannot be included in profile specific files or documents activated by spring.config.activate.on-profile.

For example, the second document configuration is invalid:

```
spring.profiles.active=prod
spring.config.activate.on-profile=prod
spring.profiles.active=metrics
```

19.6.2 Adding Active Profiles

The spring.profiles.active property follows the same ordering rules as other properties: The highest PropertySource wins. This means that you can specify active profiles in application.properties and then replace them by using the command line switch.

Sometimes, it is useful to have properties that add to the active profiles rather than replace them. The `spring.profiles.include` property can be used to add active profiles on top of those activated by the spring.profiles.active property. The SpringApplication entry point also has a Java API for setting additional profiles. See the setAdditionalProfiles() method in SpringApplication.

For example, when an application with the following properties is run, the common and local profiles will be activated even when it runs using the –spring.profiles.active switch:

```
spring.profiles.include[0]=common
spring.profiles.include[1]=local
```

Similar to spring.profiles.active, spring.profiles.include can only be used in non-profile-specific documents. This means it cannot be included in profile specific files or documents activated by spring.config.activate.on-profile.

Profile groups, which are described in the next section can also be used to add active profiles if a given profile is active.

19.6.3 Profile Groups

Occasionally the profiles that you define and use in your application are too fine-grained and become cumbersome to use. For example, you might have proddb and prodmq profiles that you use to enable database and messaging features independently.

To help with this, Spring Boot lets you define profile groups. A profile group allows you to define a logical name for a related group of profiles.

For example, we can create a production group that consists of our proddb and prodmq profiles.

```
1    spring.profiles.group.production[0]=proddb
2    spring.profiles.group.production[1]=prodmq
```

Our application can now be started using –spring.profiles.active=production to activate the production, proddb and prodmq profiles in one hit.

Similar to spring.profiles.active and spring.profiles.include, spring.profiles.group can only be used in non-profile-specific documents. This means it cannot be included in profile specific files or documents activated by spring.config.activate.on-profile.

19.6.4 Programmatically Setting Profiles

You can programmatically set active profiles by calling SpringApplication.setAdditionalProfiles(...)) before your application runs. It is also possible to activate profiles by using Spring's ConfigurableEnvironment interface.

19.6.5 Profile-specific Configuration Files

Profile-specific variants of both application.properties (or application.yaml)

19.6.6 Profile Specific Files

As well as application property files, Spring Boot will also attempt to load profile-specific files using the naming convention application-profile. For example, if your application activates a profile named prod and uses YAML files, then both application.yaml and application-prod.yaml will be considered.

Profile-specific properties are loaded from the same locations as standard application.properties, with profile-specific files always overriding the non-specific ones. If several profiles are specified, a last-wins strategy applies. For example, if profiles prod,live

are specified by the spring.profiles.active property, values in application-prod.properties can be overridden by those in application-live.properties.

In addition, files referenced through @ConfigurationProperties are loaded.

19.7 Externalized Configuration

Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use a variety of external configuration sources including Java properties files, YAML files, environment variables, and command-line arguments.

Property values can be injected directly into your beans by using the `@Value` annotation, accessed through Spring's Environment abstraction, or be bound to *structured objects* through `@ConfigurationProperties`.

Differences between both are:

- @ConfigurationProperties does support relaxed binding, allowing for more flexibility in property names and values. On the other hand, @Value does not have this feature, requiring an exact match between the property name and the field.
- @ConfigurationProperties can be validated using JSR-303 bean validation, providing a way to enforce constraints on the properties being bound. In contrast, @Value does not have built-in support for bean validation.
- @ConfigurationProperties is designed to bind properties to entire classes, allowing for a more structured and organized approach to configuration. On the other hand, @Value is typically used to bind specific properties to individual fields within a class.

Spring Boot uses a very particular PropertySource order that is designed to allow sensible overriding of values. Sources are considered in the following order:

- Default properties (specified by setting SpringApplication.setDefaultProperties).
- @PropertySource annotations on your @Configuration classes. Please note that such property sources are not added to the Environment until the application context is being refreshed. This is too late to configure certain properties such as logging.* and spring.main.* which are read before refresh begins.
- Config data (such as application.properties files).
- A RandomValuePropertySource that has properties only in random.*.
- OS environment variables.
- Java System properties (System.getProperties()).
- JNDI attributes from java:comp/env.

- ServletContext init parameters.
- ServletConfig init parameters.
- Properties from SPRING_APPLICATION_JSON (inline JSON embedded in an environment variable or system property).
- Command line arguments.
- properties attribute on your tests. Available on @SpringBootTest and the test annotations for testing a particular slice of your application.
- @DynamicPropertySource annotations in your tests.
- @TestPropertySource annotations on your tests.
- Devtools global settings properties in the $HOME/.config/spring-boot directory when devtools is active.

Config data files are considered in the following order:

- Application properties packaged inside your jar (application.properties and YAML variants).
- Profile-specific application properties packaged inside your jar (application-profile.properties and YAML variants).
- Application properties outside of your packaged jar (application.properties and YAML variants).
- Profile-specific application properties outside of your packaged jar (application-profile.properties and YAML variants).
- By default, SpringApplication converts any command line option arguments (that is, arguments starting with –, such as –server.port=9000) to a property and adds them to the Spring Environment. As mentioned previously, command line properties always take precedence over file-based property sources.

Spring Boot will automatically find and load application.properties and application.yaml files from the following locations when your application starts:

- From the classpath
- The classpath root
- The classpath /config package
- From the current directory
- The current directory

- The config/ subdirectory in the current directory
- Immediate child directories of the config/ subdirectory

The list is ordered by precedence (with values from lower items overriding earlier ones). Documents from the loaded files are added as PropertySources to the Spring Environment.

If you do not like application as the configuration file name, you can switch to another file name by specifying a spring.config.name environment property. For example, to look for myproject.properties and myproject.yaml files you can run your application as follows:

```
1    java -jar myproject.jar --spring.config.name=myproject
```

You can also refer to an explicit location by using the spring.config.location environment property. This property accepts a comma-separated list of one or more locations to check.

The following example shows how to specify two distinct files:

```
1    java -jar myproject.jar --spring.config.location=\
2    optional:classpath:/default.properties,\
3    optional:classpath:/override.properties
```

spring.config.name, spring.config.location, and spring.config.additional-location are used very early to determine which files have to be loaded. They must be defined as an environment property (typically an OS environment variable, a system property, or a command-line argument).

19.8 Spring Boot Auto-Configuration

When @EnableAutoConfiguration is present, beans annotated with @AutoConfiguration will be configured.

In spring-boot-autoconfigure.jar, /META-INF/spring/org.springframework.boot.autoconfigure.AutoConf lists the classes by default autoconfigured by Spring.

Spring's DataSourceAutoConfiguration class is one example. See fig. 19.

19.9 Testing with Spring Boot

Test support is provided by two modules: spring-boot-test contains core items, and spring-boot-test-autoconfigure supports auto-configuration for tests.

spring-boot-starter-test imports both Spring Boot test modules as well as JUnit Jupiter, AssertJ, Hamcrest, and a number of other useful libraries. Precisely:

- JUnit 5: The de-facto standard for unit testing Java applications.

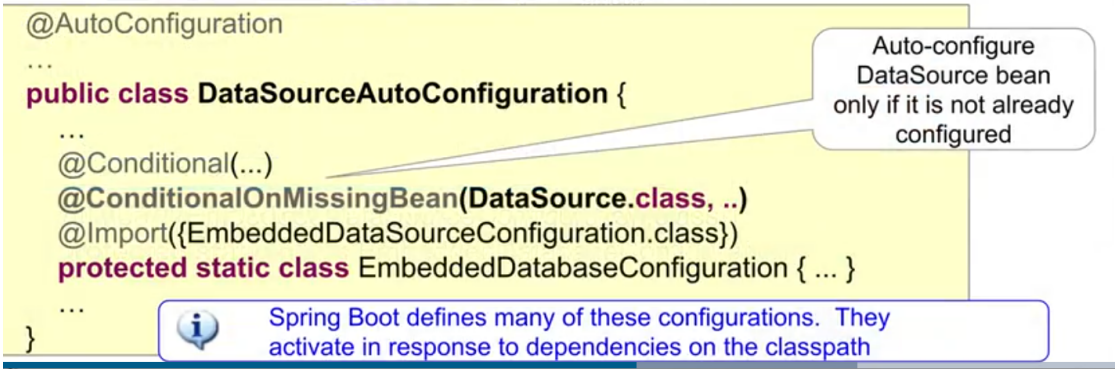


Figure 19: Spring's DataSourceAutoConfiguration class.

- Spring Test and Spring Boot Test: Utilities and integration test support for Spring Boot applications.
- AssertJ: A fluent assertion library.
- Hamcrest: A library of matcher objects (also known as constraints or predicates).
- Mockito: A Java mocking framework.
- JSONassert: An assertion library for JSON.
- JsonPath: XPath for JSON.
- Awaitility: A library for testing asynchronous systems.

By default, @SpringBootTest will not start a server. You can use the webEnvironment attribute of @SpringBootTest to further refine how your tests run:

- MOCK(Default) : Loads a web ApplicationContext and provides a mock web environment. Embedded servers are not started when using this annotation. If a web environment is not available on your classpath, this mode transparently falls back to creating a regular non-web ApplicationContext. It can be used in conjunction with @AutoConfigureMockMvc or @AutoConfigureWebTestClient for mock-based testing of your web application.
- RANDOM_PORT: Loads a WebServerApplicationContext and provides a real web environment. Embedded servers are started and listen on a random port.
- DEFINED_PORT: Loads a WebServerApplicationContext and provides a real web environment. Embedded servers are started and listen on a defined port (from your application.properties) or on the default port of 8080.
- NONE: Loads an ApplicationContext by using SpringApplication but does not provide any web environment (mock or otherwise).

19.10 Test Configuration

In Spring testing in general, we use `@ContextConfiguration(classes=...)` in order to specify which Spring @Configuration to load. When testing Spring Boot applications, this is often not required. Spring Boot's @*Test annotations search for the primary configuration automatically.

The search algorithm works up from the package that contains the test until it finds a class annotated with @SpringBootApplication or @SpringBootConfiguration.

To customize the primary configuration, one can use a *nested* `@TestConfiguration` class. Unlike a nested @Configuration class, which would be used instead of the application's primary configuration, a nested @TestConfiguration class is used *in addition to the primary configuration*.

@TestConfiguration can also be used on an *inner* class of a test or the *top-level* class to customize the primary configuration. Doing so indicates that the class should not be picked up by scanning. You can then import the class explicitly where it is required, as shown in the following example:

```
1     import org.junit.jupiter.api.Test;
2
3     import
         org.springframework.boot.test.context.SpringBootTest;
4     import org.springframework.context.annotation.Import;
5
6     @SpringBootTest
7     @Import(MyTestsConfiguration.class)
8     class MyTests {
9
10        @Test
11        void exampleTest() {
12            // ...
13        }
14
15    }
```

Note: An imported @TestConfiguration is processed earlier than an inner-class @TestConfiguration and an imported @TestConfiguration will be processed before any configuration found through component scanning.

19.11 Testing With a Mock Environment

By default, @SpringBootTest does not start the server but instead sets up a mock environment for testing web endpoints.

With Spring MVC, we can query our web endpoints using MockMvc or WebTestClient, as shown in the following example:

```
1     @SpringBootTest
2     @AutoConfigureMockMvc
3     class MyMockMvcTests {
4
5         @Test
6         void testWithMockMvc(@Autowired MockMvc mvc) throws
             Exception {
7             mvc.perform(get("/")).andExpect(status().isOk()).andExpect(cor
                 World"));
8         }
9
10        // If Spring WebFlux is on the classpath, you can
             drive MVC tests with a WebTestClient
11        @Test
12        void testWithWebTestClient(@Autowired WebTestClient
             webClient) {
13            webClient
14            .get().uri("/")
15            .exchange()
16            .expectStatus().isOk()
17            .expectBody(String.class).isEqualTo("Hello
                 World");
18        }
19
20    }
```

19.12 Auto-configured Spring MVC Tests

To test whether Spring MVC controllers are working as expected, use the @WebMvcTest annotation.

@WebMvcTest auto-configures the Spring MVC infrastructure and limits scanned beans to @Controller, @ControllerAdvice, @JsonComponent, Converter, GenericConverter, Filter, HandlerInterceptor, WebMvcConfigurer, WebMvcRegistrations, and HandlerMethodArgumentResolver.

Regular @Component and @ConfigurationProperties beans are not scanned when the @WebMvcTest annotation is used. @EnableConfigurationProperties can be used to include @ConfigurationProperties beans.

Often, @WebMvcTest is limited to a single controller and is used in combination with @MockBean to provide mock implementations for required collaborators.

@WebMvcTest also auto-configures MockMvc.

19.13 TestRestTemplate

Spring Boot also includes a TestRestTemplate that is useful in integration tests. You can get a vanilla template or one that sends Basic HTTP authentication (with a username and password). In either case, the template is fault tolerant. This means that it behaves in a test-friendly way by not throwing exceptions on 4xx and 5xx errors. Instead, such errors can be detected through the returned ResponseEntity and its status code.

TestRestTemplate can be instantiated directly in your integration tests:

```
1     import org.junit.jupiter.api.Test;
2
3     import org.springframework.boot.test.web.client.
4     TestRestTemplate;
5     import org.springframework.http.ResponseEntity;
6
7     import static
         org.assertj.core.api.Assertions.assertThat;
8
9     class MyTests {
10
11        private final TestRestTemplate template = new
             TestRestTemplate();
12
13        @Test
14        void testRequest() {
15            ResponseEntity<String> headers =
                 this.template.getForEntity(
16                "https://myhost.example.com/example",
                     String.class);
17            assertThat(
18            headers.getHeaders().getLocation()).hasHost(
19                "other.example.com");
20        }
21    }
```

Alternatively, if you use the @SpringBootTest annotation with WebEnvironment.RANDOM_PORT or WebEnvironment.DEFINED_PORT, you can inject a fully configured TestRestTemplate and start using it. If necessary, additional customizations can be applied through the RestTemplateBuilder bean. Any URLs that do not specify a host and port automatically connect to the embedded server:

```
1     import java.time.Duration;
2
3 import org.junit.jupiter.api.Test;
4
```

```
5 import
     org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.boot.test.context.SpringBootTest;
7 import org.springframework.boot.test.context.SpringBootTest.
8     WebEnvironment;
9 import
     org.springframework.boot.test.context.TestConfiguration;
10 import
     org.springframework.boot.test.web.client.TestRestTemplate;
11 import
     org.springframework.boot.web.client.RestTemplateBuilder;
12 import org.springframework.context.annotation.Bean;
13 import org.springframework.http.HttpHeaders;
14
15 import static org.assertj.core.api.Assertions.assertThat;
16
17 @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
18 class MySpringBootTests {
19
20     @Autowired
21     private TestRestTemplate template;
22
23     @Test
24     void testRequest() {
25         HttpHeaders headers =
              this.template.getForEntity("/example",
              String.class).getHeaders();
26         assertThat(headers.getLocation())
27             .hasHost("other.example.com");
28     }
29
30     @TestConfiguration(proxyBeanMethods = false)
31     static class RestTemplateBuilderConfiguration {
32
33         @Bean
34         RestTemplateBuilder restTemplateBuilder() {
35             return new
                  RestTemplateBuilder().setConnectTimeout(
36                  Duration.ofSeconds(1))
37             .setReadTimeout(Duration.ofSeconds(1));
38         }
39
40     }
41
```

```
42 }
```

19.14 Spring Boot Actuator

The recommended way to enable the features is to add a dependency on the spring-boot-starter-actuator starter.

19.14.1 Endpoints

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the health endpoint provides basic application health information.

You can enable or disable each individual endpoint and expose them (make them remotely accessible) over HTTP or JMX. An endpoint is considered to be available when it is *both enabled and exposed*.

The built-in endpoints are auto-configured only when they are available. Most applications choose exposure over HTTP, where the ID of the endpoint and a prefix of /actuator is mapped to a URL. For example, by default, the health endpoint is mapped to /actuator/health.

By default, *all endpoints except for shutdown are enabled*. To configure the enablement of an endpoint, use its management.endpoint.<id>.enabled property. The following example enables the shutdown endpoint:

```
1    management.endpoint.shutdown.enabled=true
```

If you prefer endpoint enablement to be opt-in rather than opt-out, set the `management.endpoints.enabl` property (which by default is true) to false and use individual endpoint enabled properties to opt back in. The following example enables the info endpoint and disables all other endpoints:

```
1    management.endpoints.enabled-by-default=false
2    management.endpoint.info.enabled=true
```

Disabled endpoints are removed entirely from the application context. If you want to change only the technologies over which an endpoint is exposed, use the include and exclude properties instead.

By default, only the health endpoint is *exposed* over HTTP and JMX.

To change which endpoints are exposed, use the following technology-specific include and exclude properties: management.endpoints.jmx.exposure.exclude, management.endpoints.jmx.exposure.include; management.endpoints.web.exposure.exclude, management.endpoints.web.exposure.include.

The include property lists the IDs of the endpoints that are exposed. The exclude property lists the IDs of the endpoints that should not be exposed. The exclude property takes precedence over the include property. You can configure both the include and the exclude properties with a list of endpoint IDs.

For example, to only expose the health and info endpoints over JMX, use the following property:

```
management.endpoints.jmx.exposure.include=health,info
```

* can be used to select all endpoints. For example, to expose everything over HTTP except the env and beans endpoints, use the following properties:

```
management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.exclude=env,beans
```

Monitoring and Management Over HTTP Customization:

```
// base for all endpoints
management.endpoints.web.base-path=

// for a specific endpoint
management.endpoints.web.path-mapping.<actuator>
// e.g.
management.endpoints.web.path-mapping.health=custom-health

// prt
management.server.port=8081
```

Health information Information exposed by the health endpoint depends on the `management.endpoint.he` and `management.endpoint.health.show-components` properties, which can be configured with one of the following values: never, when-authorized, always .

The default value is never. A user is considered to be authorized when they are in one or more of the endpoint's roles. If the endpoint has no configured roles (the default), all authenticated users are considered to be authorized. You can configure the roles by using the management.endpoint.health.roles property.

Health information is collected from the content of a HealthContributorRegistry (by default, all HealthContributor instances defined in your ApplicationContext). Spring Boot includes a number of auto-configured HealthContributors, and you can also write your own.

A `HealthContributor` can be either a `HealthIndicator` or a `CompositeHealthContributor`.

By default, the final system health is derived by a StatusAggregator, which sorts the statuses from each HealthIndicator based on an ordered list of statuses. The first status in the sorted list is used as the overall health status. If no HealthIndicator returns a status that is known to the StatusAggregator, an UNKNOWN status is used.

```
    cassandra
    couchbase
    db
    diskspace
    elasticsearch
    hazelcast
    influxdb
    jms
    ldap
    mail
    mongo
    neo4j
    ping
    rabbit
    redis
```

Writing Custom HealthIndicators To provide custom health information, you can register Spring beans that implement the HealthIndicator interface. You need to provide an implementation of the `health()` method and return a `Health` response. The Health response should include a status and can optionally include additional details to be displayed. The following code shows a sample HealthIndicator implementation:

```
    import org.springframework.boot.actuate.health.Health;
    import
       org.springframework.boot.actuate.health.HealthIndicator;
    import org.springframework.stereotype.Component;

    @Component
    public class MyHealthIndicator implements
       HealthIndicator {

        @Override
        public Health health() {
            int errorCode = check();
            if (errorCode != 0) {
```

```
12                  return Health.down().withDetail("Error
                       Code", errorCode).build();
13              }
14              return Health.up().build();
15          }
16
17          private int check() {
18              // perform some specific health check
19              return ...
20          }
21
22      }
```

The identifier for a given HealthIndicator is the name of the bean without the HealthIndicator suffix, if it exists. In the preceding example, the health information is available in an entry named my.

Info Endpoint: Application Information Application information exposes various information collected from all `InfoContributor` beans defined in your ApplicationContext. Spring Boot includes a number of auto-configured InfoContributor beans, and you can write your own.

Auto-configured InfoContributors may be:

```
1       // if there's a META-INF/build-info.properties resource
2       build
3       //  Exposes any property from the Environment whose
           name starts with info
4       env
5       // git.properties resource exists
6       git
7       java
8       os
9       process
```

Whether an individual contributor is enabled is controlled by its management.info.someid.enabled property.

Different contributors have different defaults for this property, depending on their prerequisites and the nature of the information that they expose.

With no prerequisites to indicate that they should be enabled, the env, java, os, and process contributors are disabled by default.

The build and git info contributors are enabled by default.

The Health Indicator status severity order be changed using the `management.health.status.order` property.

Custom Application Information When the env contributor is enabled, you can customize the data exposed by the info endpoint by setting info.* Spring properties. *All Environment properties under the info key are automatically exposed.* For example, you could add the following settings to your application.properties file:

```
info.app.encoding=UTF-8
info.app.java.source=17
info.app.java.target=17
```

Writing Custom InfoContributors To provide custom application information, you can register Spring beans that implement the InfoContributor interface.

```
import java.util.Collections;

import org.springframework.boot.actuate.info.Info;
import
   org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

@Component
public class MyInfoContributor implements
   InfoContributor {

    @Override
    public void contribute(Info.Builder builder) {
        builder.withDetail("example",
           Collections.singletonMap("key", "value"));
    }

}
```