

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 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,
13     // GenericWebApplicationContext,
14     // GenericXmlApplicationContext
15     GenericApplicationContext
16
17     GroovyWebApplicationContext
18
19     ResourceAdapterApplicationContext
20
21     StaticApplicationContext
22     StaticWebApplicationContext
23
24     XmlWebApplicationContext

```

It is possible for applications to have multiple application contexts, arranged in a parent-child hierarchy.

2.1 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, it is wrapped in an event.

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:

```
1  public class BlockedListEvent extends ApplicationEvent {  
2  
3      private final String address;  
4      private final String content;  
5  
6      public BlockedListEvent(Object source, String address,  
7          String content) {  
8          super(source);  
9          this.address = address;  
10         this.content = content;  
11     }  
12     // accessor and other methods...  
13 }
```

To publish a custom ApplicationEvent, call the publishEvent() method on an ApplicationEventPublisherAware. 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:

```

1  public class EmailService implements
2      ApplicationEventPublisherAware {
3
4      private List<String> blockedList;
5      private ApplicationEventPublisher publisher;
6
7      public void setBlockedList(List<String>
8          blockedList) {
9          this.blockedList = blockedList;
10     }
11
12     public void
13         setApplicationEventPublisher(ApplicationEventPublisher
14             publisher) {
15             this.publisher = publisher;
16         }
17
18     public void sendEmail(String address, String
19         content) {
20         if (blockedList.contains(address)) {
21             publisher.publishEvent(new
22                 BlockedListEvent(this, address,
23                     content));
24             return;
25         }
26         // send email...
27     }
28 }
```

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:

```

1  public class BlockedListNotifier implements
2      ApplicationListener<BlockedListEvent> {
3
4      private String notificationAddress;
5
6      public void setNotificationAddress(String
7          notificationAddress) {
```

```

6         this.notificationAddress = notificationAddress;
7     }
8
9     public void onApplicationEvent(BlockedListEvent
10        event) {
11         // notify appropriate parties via
12         // notificationAddress...
13     }

```

Notice that ApplicationListener is generically parameterized with the type of your custom event (BlockedListEvent in the preceding example).

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.

2.2 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
6          notificationAddress) {
7          this.notificationAddress = notificationAddress;
8      }
9
10     @EventListener
11     public void processBlockedListEvent(BlockedListEvent
12        event) {
13         // notify appropriate parties via
14         // notificationAddress...
15     }
16 }

```

2.3 The BeanFactory API

The BeanFactory API provides the underlying basis for Spring's IoC functionality. Its specific contracts are mostly used in integration with other parts of Spring and related

third-party frameworks, and its `DefaultListableBeanFactory` implementation is a key delegate within the higher-level `GenericApplicationContext` container.

`BeanFactory` and related interfaces (such as `BeanFactoryAware`, `InitializingBean`, `DisposableBean`) are important integration points for other framework components. By not requiring any annotations or even reflection, they allow for very efficient interaction between the container and its components. Application-level beans may use the same callback interfaces but typically prefer declarative dependency injection instead, either through annotations or through programmatic configuration.

2.3.1 BeanFactory or ApplicationContext?

Because an `ApplicationContext` includes all the functionality of a `BeanFactory`, it is generally recommended over a plain `BeanFactory`, except for scenarios where full control over bean processing is needed.

For many extended container features, such as annotation processing and AOP proxying, the `BeanPostProcessor` extension point is essential. If you use only a plain `DefaultListableBeanFactory`, such post-processors do not get detected and activated by default.

The following table lists features provided by the `BeanFactory` and `ApplicationContext` interfaces and implementations.

Feature	BeanFactory	ApplicationContext
Bean instantiation/wiring	Y	Y
Integrated lifecycle management	N	Y
Automatic <code>BeanPostProcessor</code> registration	N	Y
Automatic <code>BeanFactoryPostProcessor</code> registration	N	Y
Convenient <code>MessageSource</code> access	N	Y
Built-in <code>ApplicationEvent</code> publication mechanism	N	Y

To explicitly register a bean post-processor with a `DefaultListableBeanFactory`, you need to programmatically call `addBeanPostProcessor`, as the following example shows:

```
1  DefaultListableBeanFactory factory = new
   DefaultListableBeanFactory();
2  // populate the factory with bean definitions
3
4  // now register any needed BeanPostProcessor instances
5  factory.addBeanPostProcessor(new
   AutowiredAnnotationBeanPostProcessor());
6  factory.addBeanPostProcessor(new MyBeanPostProcessor());
7
8  // now start using the factory
```

To apply a BeanFactoryPostProcessor to a plain DefaultListableBeanFactory, you need to call its postProcessBeanFactory method, as the following example shows:

```
1  DefaultListableBeanFactory factory = new
2      DefaultListableBeanFactory();
3  XmlBeanDefinitionReader reader = new
4      XmlBeanDefinitionReader(factory);
5  reader.loadBeanDefinitions(new
6      FileSystemResource("beans.xml"));
7
8  // bring in some property values from a Properties file
9  PropertySourcesPlaceholderConfigurer cfg = new
10     PropertySourcesPlaceholderConfigurer();
11    cfg.setLocation(new
12        FileSystemResource("jdbc.properties"));
13
14 // now actually do the replacement
15 cfg.postProcessBeanFactory(factory);
```

In both cases, the explicit registration steps are inconvenient, which is why the various ApplicationContext variants are preferred over a plain DefaultListableBeanFactory in Spring-backed applications, especially when relying on BeanFactoryPostProcessor and BeanPostProcessor instances for extended container functionality in a typical enterprise setup.

An AnnotationConfigApplicationContext has all common annotation post-processors registered and may bring in additional processors underneath the covers through configuration annotations, such as @EnableTransactionManagement. At the abstraction level of Spring's annotation-based configuration model, the notion of bean post-processors becomes a mere internal container detail.

2.4 Overriding Beans

Bean overriding occurs when a bean is registered using an identifier that is already allocated. While bean overriding is possible, it makes the configuration harder to read. Bean overriding will be deprecated in a future release.

To disable bean overriding altogether, you can set the `allowBeanDefinitionOverriding` flag to false on the ApplicationContext before it is refreshed. In such a setup, an exception is thrown if bean overriding is used.

By default, the container logs every attempt to override a bean at INFO level so that you can adapt your configuration accordingly. While not recommended, you can silence those logs by setting the `allowBeanDefinitionOverriding` flag to true.

If you use Java Configuration, a corresponding `@Bean` method always silently overrides a scanned bean class with the same component name as long as the return type of the

@Bean method matches that bean class. This simply means that the container will call the @Bean factory method in favor of any pre-declared constructor on the bean class.

Note: *Spring Boot 2.1 disabled bean overriding by default as a defensive approach*. On respective attempts, a `BeanDefinitionOverrideException` is thrown.

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"
4         ref="item1" />
5     </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 - not, though, if a field is *final*. Field-based injection is not recommended - e.g., it makes testing harder.

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

3.4 Configuration: Implicit vs. Explicit

Also referred to as Java-based (decoupled) and annotation-based.
with both types, bean naming works differently - see 6.

3.4.1 Java-based

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

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

```
5     }
```

3.4.2 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 {}
```

4 Bean Lifecycle

4.1 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.

4.1.1 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.

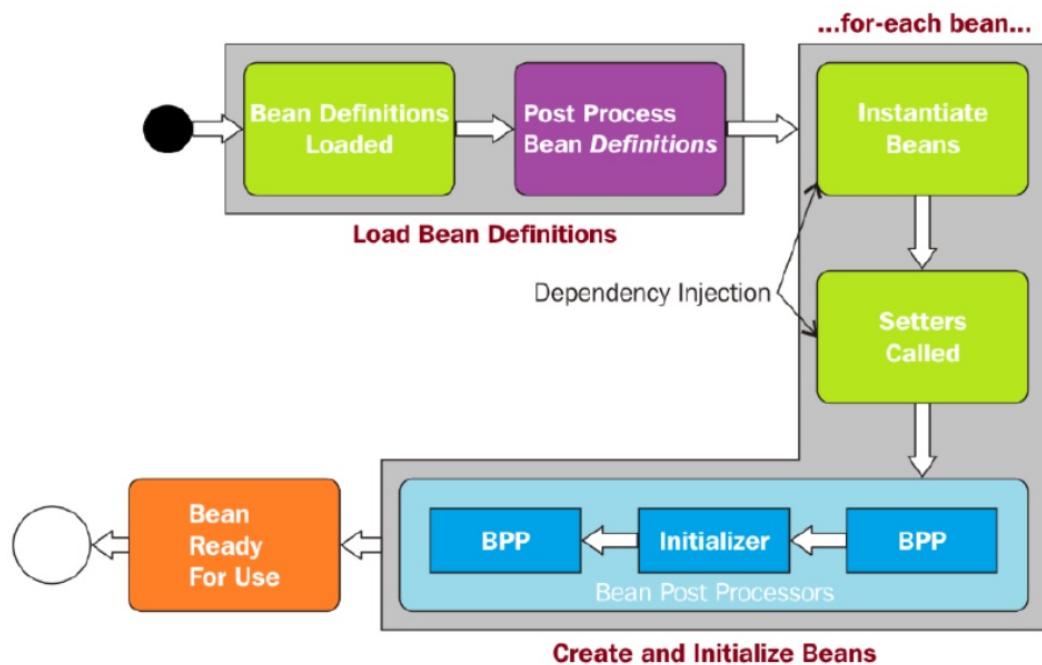


Figure 1: Lifecycle overview

Configuration Lifecycle

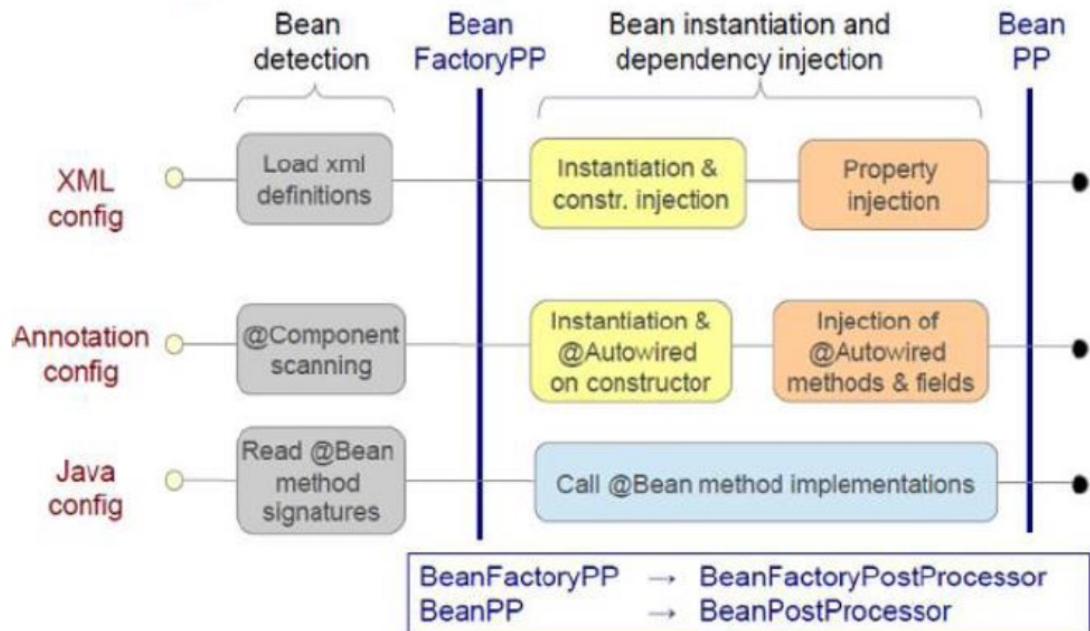


Figure 2: Zooming in on pre-instantiation

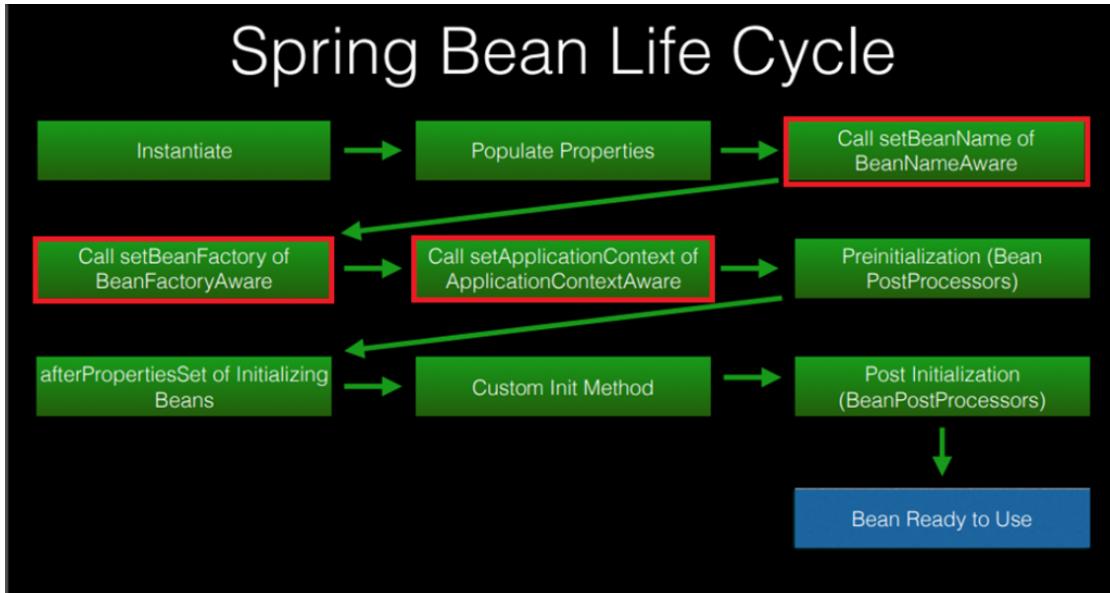


Figure 3: Zooming in on post-instantiation

4.1.2 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.

4.1.3 Instantiate beans

Injects values and bean references into beans' properties.

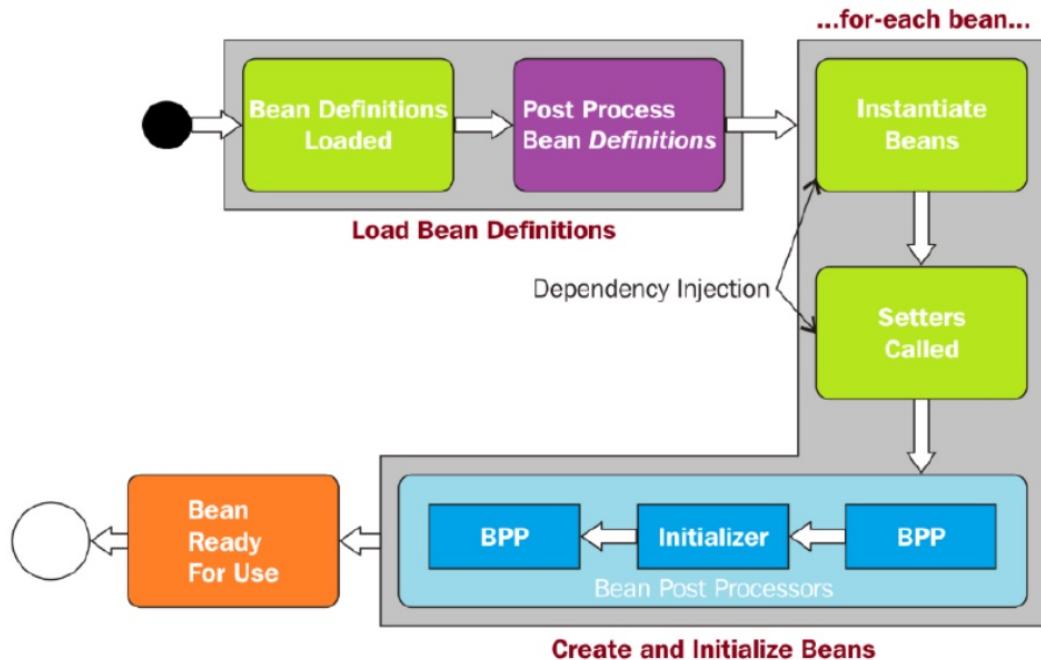


Figure 4:

- 4.1.4 Call **BeanNameAware's setBeanName()** for each bean implementing it
- 4.1.5 Call **BeanFactoryAware's setBeanFactory()** passing the bean factory for each bean implementing it
- 4.1.6 Call **ApplicationContextAware's setApplicationContext** for each bean implementing it
- 4.1.7 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
5       * new bean instance before any bean's
6       * initialization callbacks (like
7       * InitializingBean's afterPropertiesSet
8       * or a custom init-method).
9       */
10      @Nullable

```

```

8     default Object
9         postProcessBeforeInitialization(Object bean,
10            String beanName) throws BeansException {
11             return bean;
12         }
13
14        /**
15         * Apply this {@code BeanPostProcessor} to the given
16         new bean instance after any bean initialization
17         callbacks (like InitializingBean's
18         afterPropertiesSet
19         * or a custom init-method).
20         */
21     @Nullable
22     default Object
23         postProcessAfterInitialization(Object bean,
24            String beanName) throws BeansException {
25             return bean;
26         }
27     }

```

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

Figure 5 shows a no-op implementation.

4.1.8 Initialization hook 1: JSR 250's `@PostConstruct`

Of the three ways to tune in to Spring's lifecycle events, the first method called is JSR 250's `@PostConstruct` annotation.

Example:

```

1     @PostConstruct
2     public void init(){
3         System.out.println("MyService init method called");
4     }

```

`@PostConstruct` (as well as `init-method`) is 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.

Example: CustomBeanPostProcessor

```
@Component ← Can be found by component-scanner, like any other bean
public class CustomBeanPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String beanName) {
        // Some code
        return bean; // Remember to return your bean or you'll lose it!
    }

    public Object postProcessAfterInitialization(Object bean, String beanName) {
        // Some code
        return bean; // Remember to return your bean or you'll lose it!
    }
}
```

VMWare

Confidential | ©2021 VMware, Inc.

20

Figure 5: Custom bean postprocessor

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

Annotated methods can have any visibility but must take no parameters and may only return void.

4.1.9 Initialization hook 2: 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
5       * has set all bean properties.
6       * This method allows the bean instance to perform
7       * validation of its overall configuration and
```

```

    final initialization when all bean properties
    have been set.

6     */
7     void afterPropertiesSet() throws Exception;
8
9 }
```

4.1.10 Initialization hook 3: Init-Method

As last of the three, the init-method of the bean tag (the initMethod attribute of the @Bean annotation, respectively) is called.

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).

4.1.11 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.

4.1.12 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. (Annotated methods can have any visibility but must take no parameters and may only return void.)
- Any destroy method in a bean 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.

4.2 Customizing the Nature of a Bean

4.2.1 Lifecycle Callbacks

To interact with the container's management of the bean lifecycle, you can implement the `InitializingBean` and `DisposableBean` interfaces. The container calls `afterPropertiesSet()` for the former and `destroy()` for the latter to let the bean perform certain actions upon initialization and destruction of your beans.

The JSR-250 `@PostConstruct` and `@PreDestroy` annotations are generally considered best practice for receiving lifecycle callbacks in a modern Spring application. Using these annotations means that your beans are not coupled to Spring-specific interfaces.

If you do not want to use the JSR-250 annotations but you still want to remove coupling, consider init-method and destroy-method bean definition metadata.

Multiple lifecycle mechanisms configured for the same bean, with different initialization methods, are called as follows:

1. Methods annotated with `@PostConstruct`
2. `afterPropertiesSet()` as defined by the `InitializingBean` callback interface
3. A custom configured `init()` method

Destroy methods are called in the same order:

1. Methods annotated with `@PreDestroy`
2. `destroy()` as defined by the `DisposableBean` callback interface
3. A custom configured `destroy()` method

Internally, the Spring Framework uses `BeanPostProcessor` implementations to process any callback interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle behavior Spring does not by default offer, you can implement a `BeanPostProcessor` yourself.

4.2.2 Listening to Application Context startup/shutdown

In addition to the individual bean initialization and destruction callbacks, Spring-managed objects may also implement the `Lifecycle` interface so that those objects can participate in the startup and shutdown process, as driven by the container's own lifecycle.

The `Lifecycle` interface defines the essential methods for any object that has its own lifecycle requirements (such as starting and stopping some background process):

```

1  public interface Lifecycle {
2      void start();
3      void stop();
4      boolean isRunning();
5  }

```

Any Spring-managed object may implement the Lifecycle interface. Then, when the ApplicationContext itself receives start and stop signals (for example, for a stop/restart scenario at runtime), it cascades those calls to all Lifecycle implementations defined within that context. It does this by delegating to a LifecycleProcessor, shown in the following listing:

```

1  public interface LifecycleProcessor extends Lifecycle {
2      void onRefresh();
3      void onClose();
4  }

```

Notice that the LifecycleProcessor is itself an extension of the Lifecycle interface. It also adds two other methods for reacting to the context being refreshed and closed.

4.2.3 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.

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);

```

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)
5         throws BeansException {
6         this.beanFactory = beanFactory;
7     }
8     public void getMyBeanName() {
9         MyBeanName myBeanName =
10            beanFactory.getBean(MyBeanName.class);
11            System.out.println(beanFactory.isSingleton("myCustomBeanName"));
12        }
13        MyBeanFactory myBeanFactory =
14            context.getBean(MyBeanFactory.class);
15        myBeanFactory.getMyBeanName();

```

```

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

```

4.2.4 Customizing Beans by Using a BeanPostProcessor

4.2.5 Customizing Configuration Metadata with a BeanFactoryPostProcessor

4.2.6 Customizing Instantiation Logic with a FactoryBean

5 Annotations

5.1 Annotations for dependency injection

5.1.1 @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:

```

1      // constructor injection
2      @Autowired
3      public MyBean(X x){}
4
5      // method injection
6      @Autowired
7      public setX(X x){}
8
9      // field injection
10     @Autowired
11     // private works!
12     private X x;

```

5.1.2 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.

```

1
2      @Configuration
3      public class DataSourceConfig {
4
5          @Bean
6          @Primary
7          @ConfigurationProperties(prefix =
8              "spring.datasource")
9          public DataSource primaryDataSource() {
10              return DataSourceBuilder.create().build();
11          }
12
13          @Bean
14          @ConfigurationProperties(prefix =
15              "spring.second-datasource")
16          public DataSource secondaryDataSource() {
17              return DataSourceBuilder.create().build();
18          }
19      }

```

5.1.3 @Bean

@Bean marks a factory method which instantiates a Spring bean.

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

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

5.1.4 @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=AnnotationConfigApplicationContextLoader.class,
12     classes= MyAppContext.class)
13 public class Xxx {
14     @Resource(name="namedFile")
15     private File defaultFile;
16 }
```

5.1.5 @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.

```
1  public class MyApplicationContext {
2      @Bean
3          // no bean name specified - method name is used
4          public File getSomeFile() {
5              File namedFile = new File("namedFile.txt");
6              return namedFile;
7          }
8      }
9
10     @ContextConfiguration(
11         loader=AnnotationConfigApplicationContextLoader.class,
12         classes= MyApplicationContext.class)
13     public class Xxx {
14         @Inject
15         private File defaultFile;
16     }
```

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:

```
1  import jakarta.inject.Inject;
2  import jakarta.inject.Provider;
3
4  public class SimpleMovieLister {
5
6      private Provider<MovieFinder> movieFinder;
7
8      @Inject
9      public void setMovieFinder(Provider<MovieFinder>
10          movieFinder) {
11          this.movieFinder = movieFinder;
12      }
13
14      public void listMovies() {
15          this.movieFinder.get().findMovies(...);
16          // ...
17      }
18  }
```

```
17 }
```

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")
10         MovieFinder movieFinder) {
11         this.movieFinder = movieFinder;
12     }
13
14     // ...
15 }
```

5.1.6 `@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")
6     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:

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

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

```
1
2 @Configuration
3 @ComponentScan(basePackages = "org.example")
4 public class AppConfig {
5     // ...
6 }
```

Limitations of using `@Inject` and `@Named` / `@ManagedBean` are (Spring equivalent is also shown):

5.1.7 `@Value`

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

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

Spring	jakarta.inject.* Comments	
@Autowired	@Inject	@Inject has no 'required' attribute. Can be used with Java 8's Optional instead.
@Component	@Named / @ManagedBean	JSR-330 does not provide a composable model, only a way to identify named components.
@Scope("singleton")	@Singleton	The JSR-330 default scope is like Spring's prototype. However, in order to keep it consistent with Spring's general defaults, a JSR-330 bean declared in the Spring container is a singleton by default. In order to use a scope other than singleton, you should use Spring's @Scope annotation. jakarta.inject also provides a jakarta.inject.Scope annotation: however, this one is only intended to be used for creating custom annotations.
@Qualifier	@Qualifier / @Named	jakarta.inject.Qualifier is just a meta-annotation for building custom qualifiers. Concrete String qualifiers (like Spring's @Qualifier with a value) can be associated through jakarta.inject.Named.
@Value	-	-
@Lazy	-	-
ObjectFactory	Primary	jakarta.inject.Provider is a direct alternative to Spring's ObjectFactory, only with a shorter get() method name. It can also be used in combination with Spring's @Autowired or with non-annotated constructors and setter methods.

3 }

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 }
```

5.1.8 @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 }
```

5.1.9 @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 }
```

@Lazy may also be used to annotate test classes.

5.1.10 @Scope

The Spring Framework supports six scopes, four of which are available only if you use a web-aware ApplicationContext. You can also create a custom scope.

These are:

- singleton
- prototype
- request
- session
- application: Scopes a single bean definition to the lifecycle of a ServletContext.
- websocket

5.2 Context Configuration Annotations

5.2.1 @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})
```

5.2.2 @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 {}
```

5.2.3 @PropertySource

With this annotation, we set properties using a POJO.

```
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::

5.3 Bean annotations

5.3.1 @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: !, \&, |.

If a given profile is prefixed with the NOT operator (!), the annotated component will be registered if the profile is not active.

For example, given @Profile("p1", "!p2"), registration will occur if profile 'p1' is active *or* if profile 'p2' is not active. In consequence, the following will result in the profile(s) being registered:

```
1     p1 , p2 // !
2
3     p1
4
5     p3
```

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
8             {}
```

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",
2                         "embedded.jpa");
```

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

5.3.2 Conditional

Indicates that a component is only eligible for registration when all specified conditions match.

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

- as a type-level annotation on any class directly or indirectly annotated with @Component, including @Configuration classes
- as a meta-annotation, for the purpose of composing custom stereotype annotations
- as a method-level annotation on any @Bean method

If a @Configuration class is marked with @Conditional, all of the @Bean methods, @Import annotations, and @ComponentScan annotations associated with that class will be subject to the conditions.

5.3.3 @ComponentScan

The @ComponentScan annotation is used on a class annotated with @Configuration. (Equivalently, in Spring Boot, `scanBasePackageClasses` may be specified on the @SpringBootApplication class.)

@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 =
3      "com.baeldung.annotations")
4  class VehicleFactoryConfig {}
```

Or else, specifying classes:

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

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

```
1     @ComponentScan(basePackages =
2         "springapp.animals;springapp.flowers")
3     @ComponentScan(basePackages =
4         "animals,springapp.flowers")
5     @ComponentScan(basePackages = "springapp.animals
6         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 =
3             FilterType.ASSIGNABLE_TYPE, value = Rose.class))
```

5.3.4 @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.

5.3.5 @Repository

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

5.3.6 @Service

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

5.3.7 @Controller

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

5.3.8 @Configuration

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

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

5.3.9 @JdkProxyHint

In Spring Boot 2.5, the `@JdkProxyHint` annotation was introduced to address compatibility issues with GraalVM's native image generation. GraalVM is a high-performance runtime that can compile Java applications into native executables, providing improved startup time and reduced memory footprint.

When using Spring Boot with GraalVM native image generation, the default dynamic proxy mechanism based on JDK proxies may cause issues. The `@JdkProxyHint` annotation allows developers to configure specific beans or bean types to be processed differently when generating the native image. By providing hints, Spring Boot can switch to alternative proxying mechanisms, such as CGLIB proxies, to ensure compatibility with GraalVM.

5.4 Spring MVC annotations

5.4.1 @RequestMapping

This annotation can be used both at the class and at the method level.

In most cases, at the method level applications will prefer to use one of the HTTP method specific variants `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, or `@PatchMapping`.

Example, passing the path and the method (all optional):

```
1  @RequestMapping(path="/profile", method =
    RequestMethod.PUT)
```

5.5 Spring Boot Annotations

5.5.1 @SpringBootApplication

This is a combination of three annotations:

```
1  // enable registration of extra beans in the
   context or the import of additional
   configuration classes. An alternative to
   Spring's standard @Configuration that aids
   configuration detection in your integration
   tests.
2  @SpringBootApplication
3  @EnableAutoConfiguration
```

Here is its definition:

```

1      @Target(TYPE)
2      @Retention(RUNTIME)
3      @Documented
4      @Inherited
5      @SpringBootConfiguration
6      @EnableAutoConfiguration
7      @ComponentScan(excludeFilters={@Filter(
8          type=CUSTOM,
9          classes=TypeExcludeFilter.class),})
10     public @interface SpringApplication

```

5.5.2 @ConfigurationProperties

Helps keep configuration clean (see [6](#)).

This annotation has to be enabled via one of:

- @EnableConfigurationProperties on the application class

```

1      @SpringBootApplication
2      @EnableConfigurationProperties(
3          ConnectionSettings.class)
4      public class App {
5          // ...
6      }

```

- @ConfigurationPropertiesScan on the application class

```

1      @SpringBootApplication
2      @ConfigurationPropertiesScan
3      public class App {
4          // ...
5      }

```

- @Component on the configuration class

- **@ConfigurationProperties** on dedicated bean
 - Will hold the externalized properties
 - Avoids repeating the prefix
 - Data-members automatically set from corresponding properties

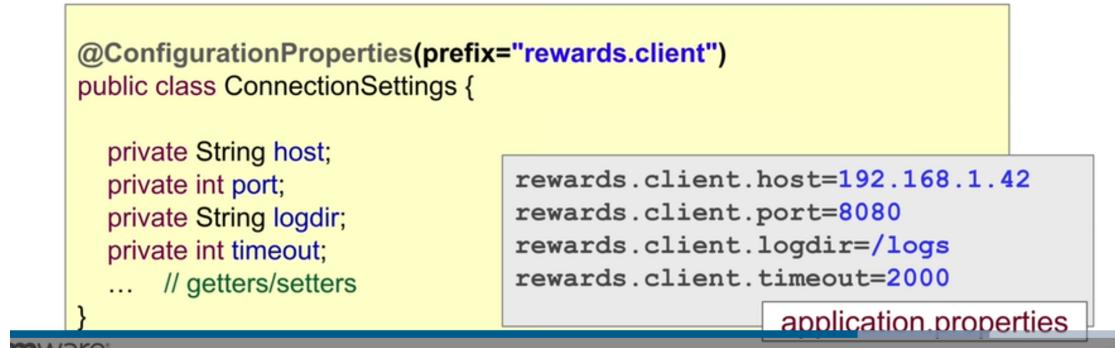


Figure 6:

```

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

```

Just like @Value, @ConfigurationProperties supports SpEL expressions.

As opposed to @Value, @ConfigurationProperties can be validated using JSR-303 bean validation, providing a way to enforce constraints on the properties being bound.

Also in contrast to @Value, @ConfigurationProperties supports relaxed binding, allowing for more flexibility in property names and values.

5.5.3 @ConditionalOnX

Determine what auto configuration does. There are:

```

1 ConditionalOnBean
2 ConditionalOnSingleCandidate
3 conditionalOnMissingBean
4
5 ConditionalOnCheckpointRestore
6
7 ConditionalOnClass
8 ConditionalOnMissingClass

```

```

9
10 ConditionalOnCloudPlatform
11
12 // allows conditional auto-configuration based on the
13 // evaluation of a SpEL expression.
13 ConditionalOnExpression
14
15 // @Conditional that matches based on the JVM version
16 // the application is running on.
16 ConditionalOnJava
17 ConditionalOnJava.Range
18
19 ConditionalOnJndi
20
21 ConditionalOnWarDeployment
22 ConditionalOnNotWarDeployment
23
24 ConditionalOnWebApplication
25 ConditionalOnWebApplication.Type
26 ConditionalOnNotWebApplication
27
28 // @Conditional that checks if the specified properties
29 // have a specific value.
29 ConditionalOnProperty
30
31 // @Conditional that only matches when the specified
32 // resources are on the classpath.
32 ConditionalOnResource
33
34 ConditionalOnThreading

```

For example, @Profile is such a condition.

Example annotation:

```

1 /*
2  When placed on a @Bean method and none of value(),
3   type(), name(), or annotation() has been specified,
4   the bean type to match defaults to the return type
5   of the @Bean method:
6
7  @Configuration
8  public class MyAutoConfiguration {
9
10    @ConditionalOnBean

```

```

8     @Bean
9         public MyService myService() {
10            ...
11        }
12    }
13 }
14
15 In the sample above the condition will match if a bean
16 of type MyService is already contained in the
17 BeanFactory.
18 */
19
20 @Conditional(
21     org.springframework.boot.autoconfigure.condition.OnBeanCondition...
22 public @interface ConditionalOnBean

```

5.5.4 RestController

Includes @Controller and @ResponseBody.

5.5.5 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
4                               userid) {}
5
6 // localhost:8080/accounts/12345
7 @GetMapping("/accounts/{accountId}")
8     public Account find (@PathVariable("accountId") long
9                           id) {}

```

@RestController Convenience

- Convenient “composed” annotation
 - Incorporates `@Controller` and `@ResponseBody`
 - Methods assumed to return REST response-data

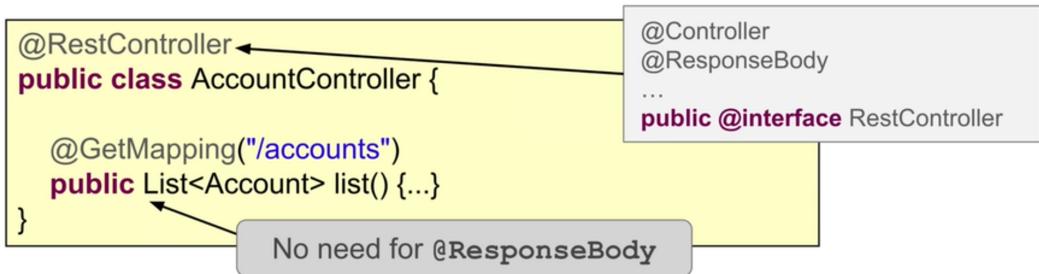


Figure 7: RestController convenience annotation.

```
8      // if argument name is missing, will take from the
9      // 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
17     // localhost:8080/accounts/12345
18     @GetMapping("/accounts/{accountId}")
19     public Account find (@PathVariable long id) {}
20
21     // localhost:8080/accounts/12345?overdrawn=true
22     @GetMapping("/accounts/{accountId}")
23     public Account find (
24         @PathVariable long accountId,
25         @RequestParam boolean overdrawn
26     ) {}
```

5.5.6 @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  
13             Posting!!!!");  
14     }  
15 }
```

5.5.7 @ResponseStatus

Used to return a status other than 200.

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

5.5.8 @RequestBody

Used to extract the request body.

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

Example:

```
1  @PostMapping("/request")
2  public ResponseEntity postController(
3      @RequestBody LoginForm loginForm) {
4
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(...){}
```

6 Bean Naming

6.1 Default Bean Naming

6.1.1 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 =
3      loggingService
4  }
```

6.1.2 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  }
```

6.2 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.

6.3 Naming Beans With @Bean and @Qualifier

6.3.1 @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  }

```

6.3.2 @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,
23                     @Qualifier("cat")Animal cat) {
24         this.dog = dog;
25         this.cat = cat;
26     }
27     public Animal getDog() {
28         return dog;
29     }
29     public Animal getCat() {

```

```

30         return cat;
31     }
32 }
```

7 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 PropertySourcePlaceholderConfigurer BeanFactoryPostProcessor prior to bean creation and can only be used in @Value annotations.
- 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 SpEL, Beans are referenced using @: e.g., `@myComponent.toString()`.

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

For example, these are two valid ways to access a system property `daily.limit`:

```
1     @Value("${daily.limit}")
2
3     @Value("#{systemProperties['daily.limit']}")
```

8 AOP in Spring

8.1 Core AOP Concepts

8.1.1 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.

8.1.2 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.

Pointcuts can be combined using &&, —— and !.

Examples:

```

1 // The execution of any public method:
2 execution(public * *(..))
3
4 // The execution of any method with a name that begins with
5 // set:
5 execution(* set*(..))
6
7 // The execution of any method defined by the
8 // AccountService interface:
8 execution(* com.xyz.service.AccountService.*(..))
9
10 // The execution of any method defined in the service
11 // package:
11 execution(* com.xyz.service.*.*(..))
12
13 //The execution of any method defined in the service
14 // 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
20 // 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)

```

8.1.3 Advice

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

- Before-advice is executed before calling the target method. It does *not* have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

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

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

```
1 @After("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(*
3     *..Repository.*(..))", throwing="e")
4 // also have to match the type of the exception
5 public void report(JoinPoint jp,
6     DataAccessException e) {
7     mailService.mailFailure(jp.getSignature(), e);
8 }
```

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

```
1 @AfterThrowing(value="execution(*
2     *..Repository.*(..))", throwing="e")
3 public void report(JoinPoint jp,
4     DataAccessException e) {
5     mailService.mailFailure(jp.getSignature(),
6         e);
7     throw new RewardsException();
8 }
```

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

8.1.4 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.

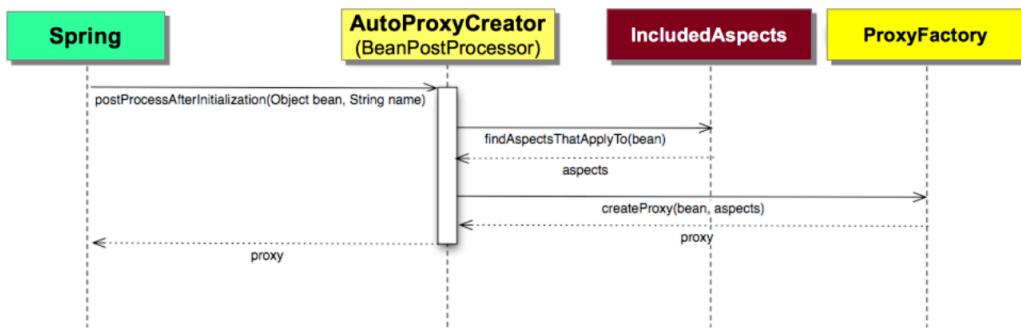
```
1 public abstract class AbstractAutoProxyCreator extends
2     ProxyProcessorSupport
```

- 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

Figure 8: Around Advice



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

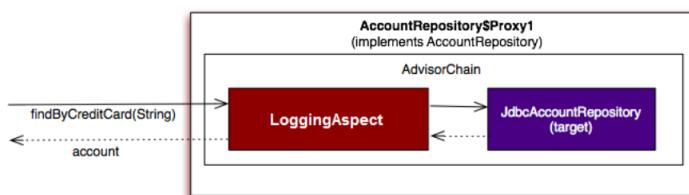


Figure 9: Proxy Creation.

Tracking Property Changes – With Context

```
@Aspect @Component
public class PropertyChangeTracker {
    private Logger logger = Logger.getLogger(getClass());

    @Before("execution(void set*(*))")
    public void trackChange(JoinPoint point) {
        String methodName = point.getSignature().getName();
        Object newValue = point.getArgs()[0];
        logger.info(methodName + " about to change to " +
                    newValue + " on " +
                    point.getTarget());
    }
}
```

JoinPoint parameter
provides context about
the intercepted point

toString() returns bean-name

Figure 10: Automatic JoinPoint injection

```
2     implements SmartInstantiationAwareBeanPostProcessor ,
3         BeanFactoryAware {
4             //...
5
6             @Override
7             public Object
8                 postProcessBeforeInstantiation(Class<?>
9                     beanClass, String beanName) {
10                Object cacheKey = getCacheKey(beanClass,
11                    beanName);
12
13                if (!StringUtils.hasLength(beanName) ||
14                    !this.targetSourcedBeans.contains(beanName))
15                {
16                    if
17                        (this.advisedBeans.containsKey(cacheKey))
18                    {
19                        return null;
20                    }
21                    if (isInfrastructureClass(beanClass) ||
22                        shouldSkip(beanClass, beanName)) {
23                        this.advisedBeans.put(cacheKey,
24                            Boolean.FALSE);
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
209
210
211
212
213
214
215
216
217
218
219
219
220
221
222
223
224
225
226
227
227
228
229
229
230
231
232
233
234
235
235
236
237
237
238
238
239
239
240
240
241
241
242
242
243
243
244
244
245
245
246
246
247
247
248
248
249
249
250
250
251
251
252
252
253
253
254
254
255
255
256
256
257
257
258
258
259
259
260
260
261
261
262
262
263
263
264
264
265
265
266
266
267
267
268
268
269
269
270
270
271
271
272
272
273
273
274
274
275
275
276
276
277
277
278
278
279
279
280
280
281
281
282
282
283
283
284
284
285
285
286
286
287
287
288
288
289
289
290
290
291
291
292
292
293
293
294
294
295
295
296
296
297
297
298
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
13
```

```

15             return null;
16         }
17     }
18 }
19
20 @Override
21 public Object
22     postProcessAfterInitialization(@Nullable Object
23         bean, String beanName) {
24     if (bean != null) {
25         Object cacheKey =
26             getCacheKey(bean.getClass(), beanName);
27         if
28             (this.earlyProxyReferences.remove(cacheKey)
29             != bean) {
30             return wrapIfNecessary(bean, beanName,
31                 cacheKey);
32         }
33     }
34     return bean;
35 }
36 }
```

8.1.5 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*.

8.2 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 (included in Spring Core). 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.

Note: Since CGLIB creates a subclass, *the class and its methods may not be final*.

8.3 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  
          this.bar();  
5      }  
6  
7      public void bar() {  
8          // some logic...  
9      }  
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  
             pojo.foo();  
18         }  
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
5             SimplePojo());
6         factory.addInterface(Pojo.class);
7         factory.addAdvice(new RetryAdvice());
8
9         Pojo pojo = (Pojo) factory.getProxy();
10        // this is a method call on the proxy!
11        pojo.foo();
12    }

```

8.4 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
2   // given target object
3   AspectJProxyFactory factory = new
4       AspectJProxyFactory(targetObject);
5
6   // add an aspect, the class must be an @AspectJ aspect
7   // you can call this as many times as you need with
8   // different aspects
9   factory.addAspect(SecurityManager.class);
10
11  // you can also add existing aspect instances, the type
12  // of the object supplied
13  // must be an @AspectJ aspect
14  factory.addAspect(usageTracker);
15
16  // now get the proxy object...
17  MyInterfaceType proxy = factory.getProxy();

```

READ_UNCOMMITTED	dirty reads
READ_COMMITTED	non-repeatable reads
REPEATABLE_READ	phantom reads
SERIALIZABLE	-

9 Transaction Management

9.1 ACID transactions

Example code:

```

1 BEGIN TRANSACTION;
2 SELECT * FROM T;
3 WAITFOR DELAY '00:01:00',
4 SELECT * FROM T;
5 COMMIT;
```

Isolation levels and what can still happen:

With READ_COMMITTED, any (committed) data that have been inserted between reads are read, possibly resulting in non-repeatable reads.

With REPEATABLE_READ, the second read is guaranteed to display at least the rows that were returned from the first read (unchanged). New rows may have been added by a concurrent transaction in the meantime, but the existing rows cannot be deleted nor changed.

With SERIALIZABLE, the first read sees exactly the same data as the second.

9.2 Propagation Mode

Note: The NESTED propagation behavior indicates that a new transaction should be created, but it will run within an existing transaction context. The nested transaction can be rolled back independently of the outer transaction. If no transaction context exists when the method is called, a new transaction will be created.

9.3 Declarative

Declarative transaction management is made possible with Spring aspect-oriented programming (AOP), and needs to be enabled using `@EnableTransactionManagement`. 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.

Enum Constant	Description
MANDATORY	Support a current transaction, throw an exception if none exists.
NESTED	Execute within a nested transaction if a current transaction exists, behave like REQUIRED otherwise.
NEVER	Execute non-transactionally, throw an exception if a transaction exists.
NOT_SUPPORTED	Execute non-transactionally, suspend the current transaction if one exists.
REQUIRED	Support a current transaction, create a new one if none exists.
REQUIRES_NEW	Create a new transaction, and suspend the current transaction if one exists.
SUPPORTS	Support a current transaction, execute non-transactionally if none exists.

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:

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.

Note: The `@Transactional` annotation in Spring is not sufficient to handle distributed transactions seamlessly across microservices. It is limited to managing transactions within a single service or database.

9.4 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
```

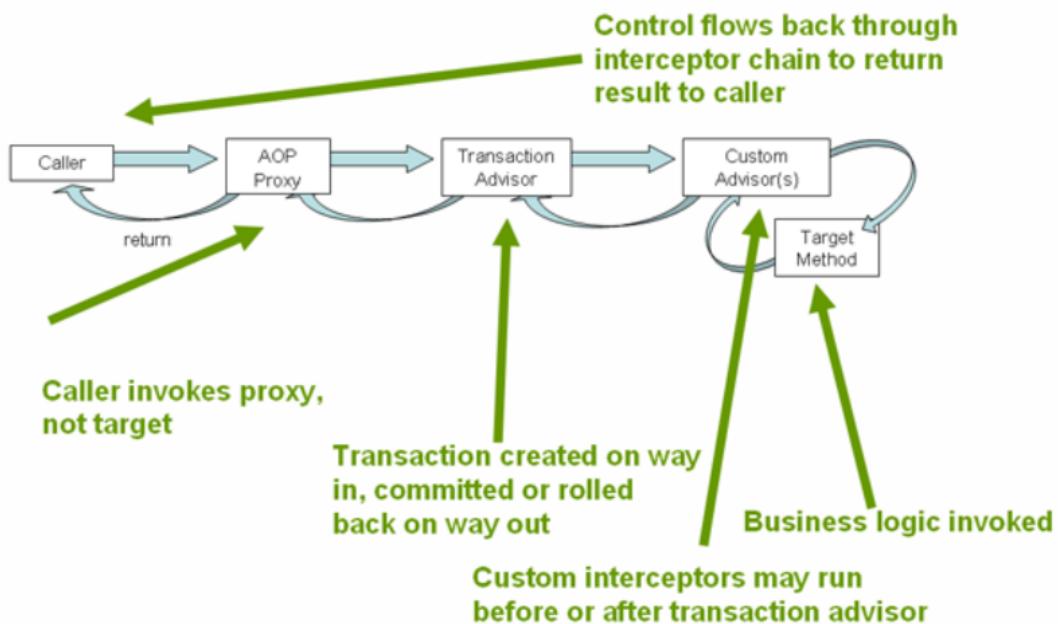


Figure 11: Spring Transaction Management Overview

```

3     @Autowired
4     private PlatformTransactionManager
5             transactionManager;
6
7     private TransactionTemplate transactionTemplate;
8
9     @BeforeEach
10    void setUp() {
11         transactionTemplate = new
12             TransactionTemplate(transactionManager);
13     }
14
15     // omitted
16 }
```

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

Note: commit before eventual rollback (for consistency)!

10 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	

10.1 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.

10.2 Using JdbcTemplate

This is the central class in the JDBC core package.

This class executes SQL queries or updates, initiating iteration over ResultSets and catching JDBC exceptions and translating them to the generic, more informative exception hierarchy defined in the org.springframework.dao package.

Because this class is parameterizable by the callback interfaces and the SQLExceptionTranslator interface, there should be no need to subclass it.

All SQL operations performed by this class are logged at debug level, using "org.springframework.jdbc.core" as log category.

Note: JdbcTemplate acquires a database connection only during a method call, which requires data access, and releases it as soon as the rows are fetched.

10.2.1 Queries

Examples:

```
1
2 Actor actor = jdbcTemplate.queryForObject(
```

```

3   "select first_name, last_name from t_actor where id =
4     ?",
5   (resultSet, rowNum) -> {
6     Actor newActor = new Actor();
7     newActor.setFirstName(
8       resultSet.getString("first_name"));
9     newActor.setLastName(
10      resultSet.getString("last_name"));
11    return newActor;
12  },
13  1212L);
14
15 List<Actor> actors = this.jdbcTemplate.query(
16   "select first_name, last_name from t_actor",
17   (resultSet, rowNum) -> {
18     Actor actor = new Actor();
19     actor.setFirstName(
20       resultSet.getString("first_name"));
21     actor.setLastName(
22       resultSet.getString("last_name"));
23     return actor;
24   });
25
26 // If the last two snippets of code actually existed in
27 // the same application, it would make sense to remove
28 // the duplication present in the two RowMapper lambda
29 // expressions and extract them out into a single field
30 // that could then be referenced by DAO methods as
31 // needed. For example, it may be better to write the
32 // preceding code snippet as follows:
33
34 private final RowMapper<Actor> actorRowMapper =
35   (resultSet, rowNum) -> {
36     Actor actor = new Actor();
37     actor.setFirstName(
38       resultSet.getString("first_name"));
39     actor.setLastName(
40       resultSet.getString("last_name"));
41     return actor;
42   };
43
44 public List<Actor> findAllActors() {
45   return this.jdbcTemplate.query("select first_name,
46     last_name from t_actor", actorRowMapper);

```

10.2.2 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));

```

10.2.3 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
3     integer, name varchar(100))");
4
5   //The following example invokes a stored procedure:
6   this.jdbcTemplate.update(
7     "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",
7     Long.valueOf(unionId));

```

10.2.4 Callback interfaces

JdbcTemplate callback (functional, i.e., implementors must implement the abstract method) interfaces are :

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

11 Data Access with JPA

In most Spring Boot applications that use JPA, you don't need to explicitly use the `@EnableJpaRepositories` annotation. By adding the `spring-data-jpa` dependency and annotating your main class with `@SpringBootApplication`, Spring Boot will automatically enable JPA repositories and scan the appropriate packages for interfaces extending `JpaRepository`.

However, if you need to customize package scanning or provide additional configuration options for JPA repositories, you can use `@EnableJpaRepositories` within your `@SpringBootApplication` class.

11.1 Repository Query Language

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

```
1 interface PersonRepository extends Repository<Person ,  
2   Long> {  
3  
4     List<Person>  
5       findByEmailAddressAndLastname(EmailAddress  
6         emailAddress , String lastname);  
7  
8     // Enables the distinct flag for the query  
9     List<Person>  
10       findDistinctPeopleByLastnameOrFirstname(String  
11         lastname , String firstname);  
12     List<Person>  
13       findPeopleDistinctByLastnameOrFirstname(String  
14         lastname , String firstname);  
15  
16     // Enabling ignoring case for an individual property  
17     List<Person> findByLastnameIgnoreCase(String  
18       lastname);  
19     // Enabling ignoring case for all suitable properties
```

```

12     List<Person>
13         findByLastnameAndFirstnameAllIgnoreCase(String
14             lastname, String firstname);
15
16     // Enabling static ORDER BY for a query
17     List<Person>
18         findByLastnameOrderByFirstnameAsc(String
19             lastname);
20     List<Person>
21         findByLastnameOrderByFirstnameDesc(String
22             lastname);
23 }

```

11.2 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
12     // Targets the pk property (the one marked with @Id
13     // which is considered to be the identifier) as it
14     // refers to a CrudRepository base repository
15     // method.
16     Optional<User> findById(Long id);
17
18     // Targets the pk property by name as it is a
19     // derived query.
20     Optional<User> findByPk(Long pk);
21
22     // Targets the id property by using the descriptive
23     // token between find and by to avoid collisions

```

```

        with reserved methods.
18     Optional<User> findUserById(Long id);
19 }
```

11.3 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
2      JpaRepository<User, Long> {
3
4      @Query("select u from User u where u.emailAddress = ?1")
5      User findByEmailAddress(String emailAddress);
6  }
```

11.4 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
2      pageable);
3
4  Slice<User> findByLastname(String lastname, Pageable
5      pageable);
6
7  List<User> findByLastname(String lastname, Sort sort);
8
9  List<User> findByLastname(String lastname, Sort sort,
10     Limit limit);
```

```
9     List<User> findByLastname(String lastname, Pageable  
    pageable);
```

11.5 Repository Query Keywords

```
1  // General query method returning typically the  
2  // repository type, a Collection or Streamable subtype  
3  // or a result wrapper such as Page, GeoResults or any  
4  // other store-specific result wrapper. Can be used as  
5  // findBy..., findMyDomainTypeBy... or in combination  
6  // with additional keywords.  
7  find...By, read...By, get...By, query...By,  
8  search...By, stream...By  
9  
10 // Exists projection, returning typically a boolean  
11 // result.  
12 exists...By  
13  
14 // Count projection returning a numeric result.  
15 count...By  
16  
17 // Delete query method returning either no result  
18 // (void) or the delete count.  
19 delete...By, remove...By  
20  
21 // Limit the query results to the first <number> of  
22 // results. This keyword can occur in any place of the  
23 // subject between find (and the other keywords) and by.  
24 ...First<number>..., ...Top<number>...  
25  
26 // Use a distinct query to return only unique results.  
27 // Consult the store-specific documentation whether  
28 // that feature is supported. This keyword can occur in  
29 // any place of the subject between find (and the other  
30 // keywords) and by.  
31 ...Distinct...
```

11.6 Accessing nested objects

Custom finders can also navigate relationships.

For example, if the repository's domain type were Person and it had an Address field with ZipCode, we could write a custom finder that uses the ZipCode of the Entity Address to filter the Person entities containing that Address.

Alternatives are:

```
1  findByAddress_ZipCode(ZipCode zipCode)
2
3  findByAddressZipCode(ZipCode zipCode)
```

11.7 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).

12 Spring Security

12.1 Overview

Fig. 12 shows a birds-eye view of how Spring Security takes care of authentication and authorization.

12.1.1 Servlet Filters and the FilterChain

Spring Security's Servlet support is based on Servlet Filters.

What happens is the following: The client sends a request to the application, and the container creates a `FilterChain`, which contains the Filter instances and Servlet that should process the `HttpServletRequest`, based on the path of the request URI.

In a Spring MVC application, the Servlet is an instance of `DispatcherServlet`. At most, one Servlet can handle a single `HttpServletRequest` and `HttpServletResponse`. However, more than one Filter can be used to:

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	GreaterThanOrEqualTo, IsGreaterThanOrEqualTo
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	LessThanOrEqualTo, IsLessThanOrEqualTo
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

- Prevent downstream Filter instances or the Servlet from being invoked. In this case, the Filter typically writes the `HttpServletResponse`.
- Modify the `HttpServletRequest` or `HttpServletResponse` used by the downstream Filter instances and the Servlet.

The power of the Filter comes from the `FilterChain` that is passed into it.

12.1.2 DelegatingFilterProxy

Spring Security provides a Filter implementation named `DelegatingFilterProxy` that bridges between the Servlet container's lifecycle and Spring's `ApplicationContext`.

The Servlet container allows registering Filter instances by using its own standards, but it is not aware of Spring-defined Beans. You can register `DelegatingFilterProxy`

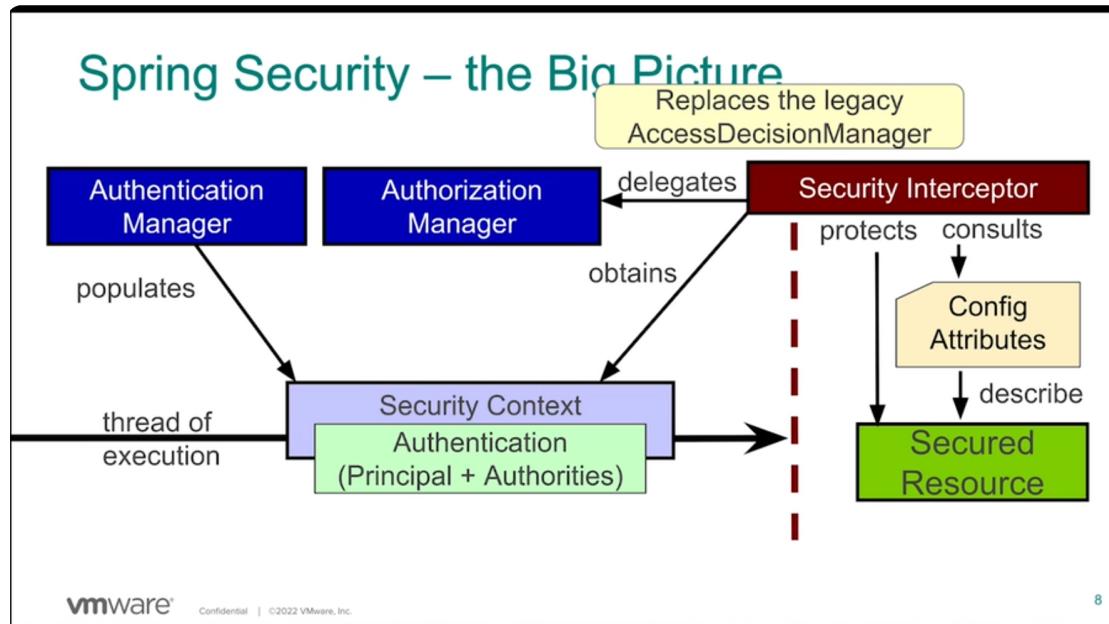


Figure 12: Spring Security Architecture Overview

through the standard Servlet container mechanisms but delegate all the work to a Spring Bean that implements Filter.

12.1.3 FilterChainProxy

Spring Security's Servlet support is contained within **FilterChainProxy**. **FilterChainProxy** is a special Filter provided by Spring Security that allows delegating to many Filter instances through **SecurityFilterChain**. *Since FilterChainProxy is a Bean, it is typically wrapped in a DelegatingFilterProxy.*

12.1.4 SecurityFilterChain

The Security Filters in **SecurityFilterChain** are typically Beans, but they are registered with **FilterChainProxy** instead of **DelegatingFilterProxy**.

If there are multiple **SecurityFilterChains**, **FilterChainProxy** decides which **SecurityFilterChain** should be used. Only the first **SecurityFilterChain** that matches is invoked.

If a URL of `/api/messages/` is requested, it first matches on the **SecurityFilterChain0** pattern of `/api/**`, so only **SecurityFilterChain0** is invoked, even though it also matches on **SecurityFilterChain1**. If a URL of `/messages/` is requested, it does not match on the **SecurityFilterChain0** pattern of `/api/**`, so **FilterChainProxy** continues trying each **SecurityFilterChain**. Assuming that no other **SecurityFilterChain** instances match, **SecurityFilterChain** is invoked.

Here (13) is an overview of the filter architecture.

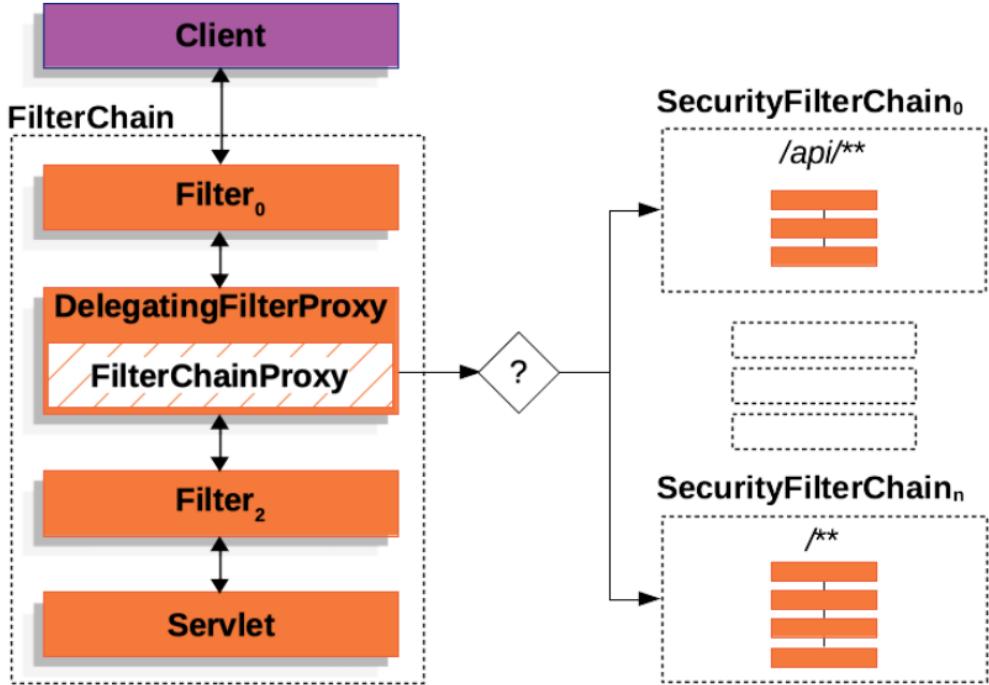


Figure 13: DelegatingFilterProxy, FilterChainProxy, and SecurityFilterChain in Spring Security

See also 14 and 15 .

12.2 Authentication

Spring Security offers the following authentication mechanisms: Username and Password, OAuth 2.0, SAML 2.0, CAS, Remember Me, JAAS Authentication, OpenID, Pre-Authentication Scenarios and X509 Authentication.

Overview (see 16):

12.3 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:

Spring Security Filter Chain – 2

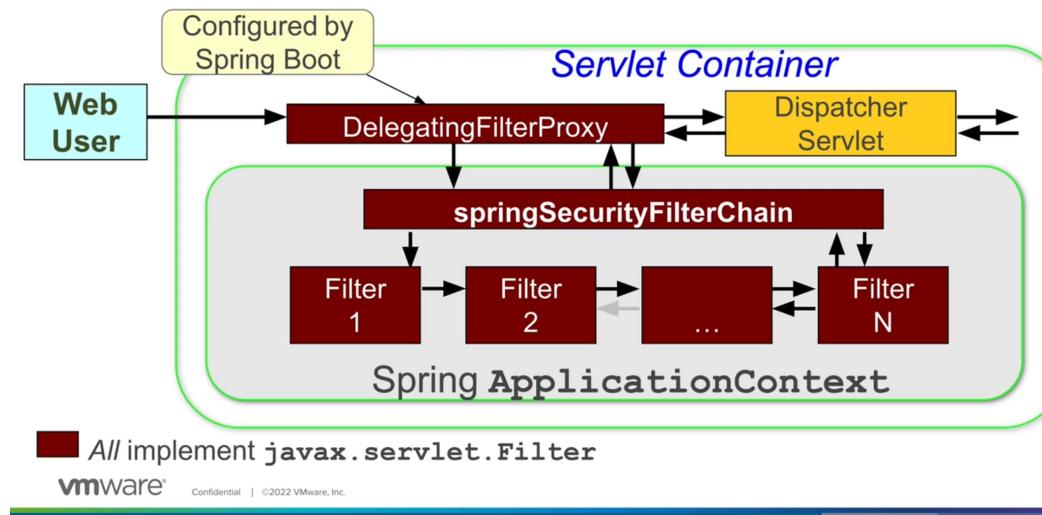


Figure 14: Spring Security Filter Chain

```
1 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
3     grantedAuthorityDefaults() {
4         return new GrantedAuthorityDefaults("MYPREFIX_");
5     }
```

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

Example Filter: SecurityContextPersistenceFilter

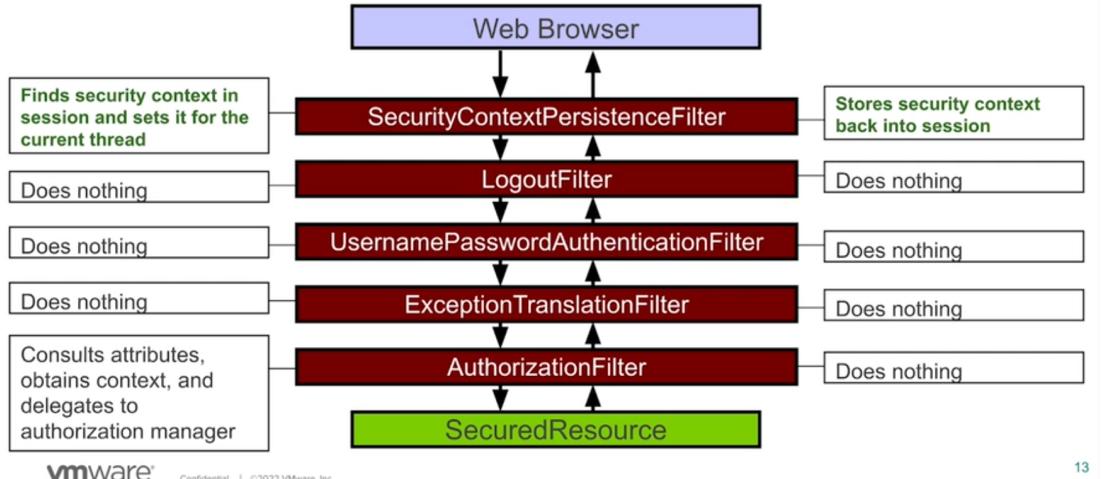


Figure 15: Example Filter Chain

12.3.1 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>
2         authentication, Object secureObject);
3
4     default void verify(Supplier<Authentication>
5         authentication, Object secureObject)
6         throws AccessDeniedException {
7         // ...
8     }
```

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.

Spring Security Authentication Flow

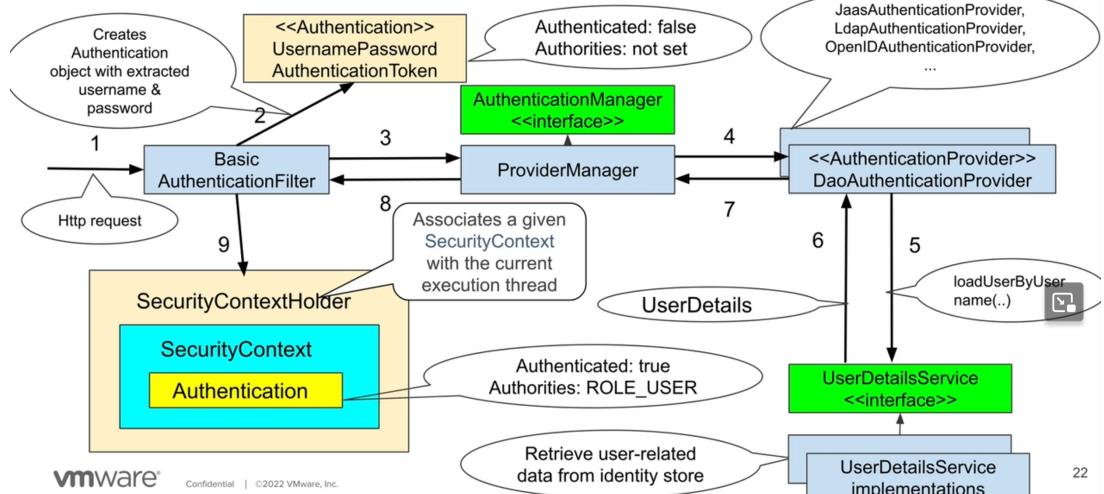


Figure 16: Spring Security Filter Chain

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 [17](#).

12.3.2 Configuring URL interception

In Spring Security, URL intercept rules are configured using `authorizeRequests()` and `requestMatchers()` methods, typically in the `HttpSecurity` configuration. When defining URL access rules, it's crucial to specify them in the right order: Most-specific rules must come first.

Example:

```

1  @Override
2  protected void configure(HttpSecurity http) throws
3      Exception {
4      http
5          .authorizeRequests()

```

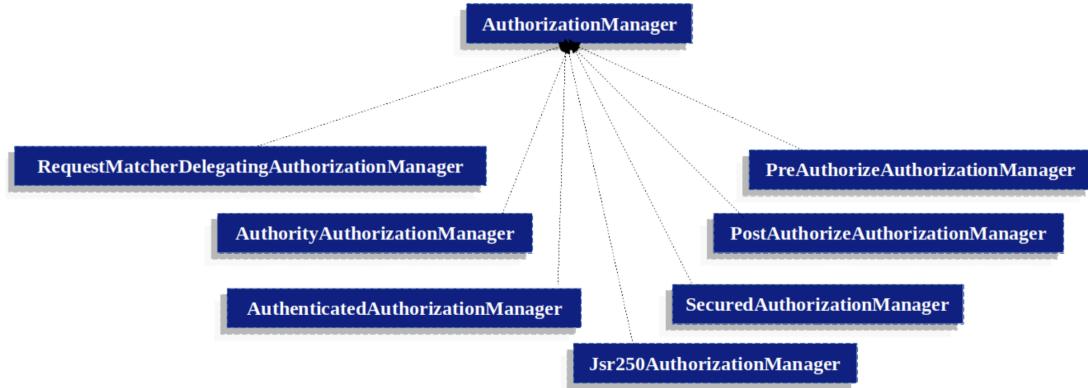


Figure 17: Spring Security Filter Chain

```

5     .requestMatchers("/admin/settings").hasRole("ADMIN")
       // Most specific
6     .requestMatchers("/admin/**").hasRole("USER")
       // Less specific
7     .requestMatchers("/**").permitAll()
       // Least specific
8     .and()
9     .formLogin();
10 }

```

12.3.3 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. This is because in the annotation interface EnableMethodSecurity, the parameter prePostEnabled is, by default, true.

Pre-Post hooks All of @PreFilter, @PostFilter, @PreAuthorize and @PostAuthorize are to be passed SpEL expressions. See interfaces in `org.springframework.security.access.prepost`:

```

1   // Annotation for specifying a method access-control
2   // expression which will be evaluated after a method
3   // has been invoked.
4   PostAuthorize
5
6   Annotation for specifying a method filtering expression
7   which will be evaluated after a method has been
8   invoked.
9   PostFilter
10
11  // Annotation for specifying a method access-control
12  // expression which will be evaluated to decide whether
13  // a method invocation is allowed or not.
14  PreAuthorize
15
16  // Annotation for specifying a method filtering
17  // expression which will be evaluated before a method
18  // has been invoked.
19  PreFilter

```

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 ==
5                      authentication.name")
6      public Customer readCustomer(String id) { ... }
7  }

```

Note: The @Secured annotation doesn't support Spring Expression Language (SpEL).

@Secured and @jsr250Enabled These parameters to EnableMethodSecurity need to be explicitly enabled.

The `@Secured` annotation is used to specify a list of roles on a method.

Example:

```
1  @Secured({ "ROLE_VIEWER", "ROLE_EDITOR" })
2  public boolean isValidUsername(String username) {
3      return userRoleRepository.isValidUsername(username);
4 }
```

`@RolesAllowed` is JSR-250's equivalent. Example:

```
1  @RolesAllowed({ "ROLE_VIEWER", "ROLE_EDITOR" })
2  public boolean isValidUsername2(String username) {
3      //...
4 }
```

13 Testing with Spring

13.1 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.

It also provides mocks for Environment and PropertySource, JNDI, and the Servlet API.

13.2 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 String separator,
9     String blockCommentStartDelimiter,
10    String blockCommentEndDelimiter
11 ) 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
2 // using SQL scripts defined in external resources.
3
4 public class ResourceDatabasePopulator implements
5     DatabasePopulator
6
7 /**
8 * Call addScript(org.springframework.core.io.Resource) to
9 * add a single SQL script location.
10 Call
11     addScripts(org.springframework.core.io.Resource...)
12     to add multiple SQL script locations.
13 Call populate(java.sql.Connection) or
14     execute(javax.sql.DataSource) to initialize or clean
15     up the database using the configured scripts.
16 */
17
```

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.

13.3 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.

13.3.1 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.

13.3.2 Static imports

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

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

13.3.3 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 =
8               MockMvcBuilders.standaloneSetup(new
9                   AccountController()).build();
10      }
11
12  }

```

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

```

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

```

13.3.4 Queries with MockMvc

Example queries using MockMvc:

```

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

```

```

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

```

14 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.

It is packaged in *spring-webmvc.jar*; packages are prefixed with *org.springframework.web*.

14.1 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
2   // session attributes, without direct use of the
3   // Servlet API.
4   WebRequest
5
6   // Choose any specific request or response type -for
7   // example, ServletRequest, HttpServletRequest, or
8   // Spring's MultipartRequest,
9   // MultipartHttpServletRequest.
10  jakarta.servlet.ServletRequest,
11      jakarta.servlet.ServletResponse
12
13  jakarta.servlet.http.HttpSession
14
15  java.security.Principal
16
17  HttpMethod

```

```

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
21 // Servlet API.
21 java.io.InputStream, java.io.Reader
22
23 // For access to the raw response body as exposed by
24 // 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,
32 // including multipart files. Parameter values are
33 // converted to the declared method argument type.
34
35 @RequestHeader
36
37
38 // For access to the HTTP request body. Body content is
39 // converted to the declared method argument type by
40 // using HttpMessageConverter implementations.
41 @RequestBody
42
43
44 // For access to a part in a multipart/form-data
45 // request, converting the part's body with an
46 // HttpMessageConverter.
@RequestPart

```

```

47   // For access to the model that is used in HTML
48   // controllers and exposed to templates as part of view
49   // rendering.
50   java.util.Map, org.springframework.ui.Model,
51   org.springframework.ui.ModelMap
52
53   // Attributes to use in case of a redirect (that is, to
54   // be appended to the query string) and flash
55   // attributes to be stored temporarily until the
56   // request after redirect.
57   RedirectAttributes
58
59   // For access to an existing attribute in the model
60   // (instantiated if not present) with data binding and
61   // validation applied. See @ModelAttribute as well as
62   // Model and DataBinder.
63   @ModelAttribute
64
65   // For access to errors from validation and data
66   // binding for a command object (that is, a
67   // @ModelAttribute argument) or errors from the
68   // validation of a @RequestBody or @RequestPart
69   // arguments.
70   Errors, BindingResult
71
72   // For preparing a URL relative to the current
73   // request's host, port, scheme, context path, and the
74   // literal part of the servlet mapping. See URI Links.
75   UriComponentsBuilder
76
77   // For access to any session attribute, in contrast to
78   // model attributes stored in the session as a result
79   // of a class-level @SessionAttributes declaration.
80   @SessionAttribute
81
82   // If a method argument is not matched to any of the
83   // earlier values in this table and it is a simple type
84   // (as determined by BeanUtils#isSimpleProperty), it is
85   // resolved as a @RequestParam. Otherwise, it is
86   // resolved as a @ModelAttribute.

```

Supported return values *include*:

```
1 // The return value is converted through
  // HttppMessageConverter 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 HttppMessageConverter implementations and written to the response.
5 ResponseEntity<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 org.springframework.ui.Model
22 java.util.Map
```

14.2 Securing Endpoints

To enable Spring Security integration with Spring MVC, add the `@EnableWebSecurity` annotation to your configuration.

Configuring HTTP security using `HttpSecurity` in a class extending `WebSecurityConfigurerAdapter` allows you to define security rules for your application, such as requiring authentication for certain endpoints.

Example:

```
1  @Configuration
2  @EnableWebSecurity
3  public class SecurityConfig extends
4      WebSecurityConfigurerAdapter {
5
6      @Override
7      protected void configure(HttpSecurity http) throws
8          Exception {
9          http
10         .authorizeRequests()
11         .antMatchers("/public/**").permitAll()
12         .anyRequest().authenticated()
13         .and()
14         .formLogin().permitAll()
15         .and()
16         .logout().permitAll();
    }
}
```

Spring Security provides deep integration with how Spring MVC matches on URLs with MvcRequestMatcher. This is helpful to ensure that your Security rules match the logic used to handle your requests. Consider a controller that is mapped as follows:

```
1  @RequestMapping("/admin")
2  public String admin() {
3      // ...
4  }
```

To restrict access to this controller method to admin users, you can provide authorization rules by matching on the HttpServletRequest with the following:

```
1  @Bean
2  public SecurityFilterChain filterChain(HttpSecurity
3      http) throws Exception {
4      http
5         .authorizeHttpRequests((authorize) -> authorize
6         .requestMatchers("/admin").hasRole("ADMIN")
7         );
8      return http.build();
}
```

With either configuration, the /admin URL requires the authenticated user to be an admin user. However, depending on our Spring MVC configuration, the /admin.html URL also maps to our admin() method. Additionally, depending on our Spring MVC configuration, the /admin URL also maps to our admin() method.

The problem is that our security rule protects only /admin. We could add additional rules for all the permutations of Spring MVC, but this would be quite verbose and tedious.

Fortunately, when using the requestMatchers DSL method, Spring Security automatically creates a MvcRequestMatcher if it detects that Spring MVC is available in the classpath. Therefore, it will protect the same URLs that Spring MVC will match on by using Spring MVC to match on the URL.

14.3 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
3      WebMvcConfigurer {
4
5      @Override
6      public void addFormatters(FormatterRegistry
7          registry) {
8          // ...
9      }
10 }
```

14.4 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
3      WebMvcConfigurer {
4
5      @Override
6      public Validator getValidator() {
7          Validator validator = new
8              OptionalValidatorFactoryBean();
9          // ...
10         return validator;
11     }
12 }
```

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  }
```

14.5 Interceptors

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

```
1  @Configuration
2  public class WebConfiguration implements
3      WebMvcConfigurer {
4
5      @Override
```

```

5     public void addInterceptors(InterceptorRegistry
6         registry) {
7             registry.addInterceptor(new
8                 LocaleChangeInterceptor());
9         }
10 }
```

14.6 Filters

Implementing the Filter interface and registering it as a bean allows you to define custom behavior for each HTTP request. The filter will be applied to all incoming requests.

Example:

```

1  @Component
2 public class RequestLoggingFilter implements Filter {
3
4     @Override
5     public void doFilter(ServletRequest request,
6         ServletResponse response, FilterChain chain)
7     throws IOException, ServletException {
8         // Log request details
9         chain.doFilter(request, response);
10    }
11 }
```

14.7 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
3      WebMvcConfigurer {
4
5      @Override
6      public void configureContentNegotiation(
7          ContentNegotiationConfigurer configurer) {
8              configurer.mediaType("json",
9                  MediaType.APPLICATION_JSON);
```

```

8         configurer.mediaType("xml",
9             MediaType.APPLICATION_XML);
10    }

```

14.8 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
3     WebMvcConfigurer {
4
5     @Override
6     public void configureMessageConverters(
7         List<HttpMessageConverter<?>> converters) {
8         Jackson2ObjectMapperBuilder builder = new
9             Jackson2ObjectMapperBuilder()
10            .indentOutput(true)
11            .dateFormat(new SimpleDateFormat("yyyy-MM-dd"))
12            .modulesToInstall(new ParameterNamesModule());
13         converters.add(new
14             MappingJackson2HttpMessageConverter(
15                 builder.build()));
16         converters.add(new
17             MappingJackson2XmlHttpMessageConverter(
18                 builder.createXmlMapper(true).build()));
19     }
20 }

```

14.9 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:

```
1  StringHttpMessageConverter
2  FormHttpMessageConverter
3  ByteArrayHttpMessageConverter
4  MarshallingHttpMessageConverter
5  JsonbHttpMessageConverter
6  MappingJackson2HttpMessageConverter
7  Jaxb2RootElementHttpMessageConverter
8  ProtobufHttpMessageConverter
9  ProtobufJsonFormatHttpMessageConverter
10 AtomFeedHttpMessageConverter
```

14.10 URI Links

14.10.1 UriComponents

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

```
1  UriComponents uriComponents = UriComponentsBuilder
2  .fromUriString("https://example.com/hotels/{hotel}")
3  .queryParam("q", "{q}")
4  .encode()
5  .build();
6
7  URI uri = uriComponents.expand("Westin", "123").toUri();
```

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

```
1  URI uri = UriComponentsBuilder
2  .fromUriString("https://example.com/hotels/{hotel}")
3  .queryParam("q", "{q}")
4  .encode()
```

```
5     .buildAndExpand("Westin", "123")
6     .toUri();
```

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

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

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

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

14.10.2 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:

```
1 // import org.springframework.web.util.
2     DefaultUriBuilderFactory.EncodingMode;
3
4 String baseUrl = "https://example.org";
5 DefaultUriBuilderFactory factory = new
6     DefaultUriBuilderFactory(baseUrl);
7 factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);
8
9 RestTemplate restTemplate = new RestTemplate();
10 restTemplate.setUriTemplateHandler(factory);
11
12 In addition, you can also use DefaultUriBuilderFactory
13 directly. It is similar to using
14 UriComponentsBuilder but, instead of static factory
```

```

    methods, it is an actual instance that holds
    configuration and preferences.

12
13     String baseUrl = "https://example.com";
14     DefaultUriBuilderFactory uriBuilderFactory = new
15         DefaultUriBuilderFactory(baseUrl);
16
17     URI uri = uriBuilderFactory.uriString("/hotels/{hotel}")
18         .queryParam("q", "{q}")
19         .build("Westin", "123");

```

14.11 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
3      WebMvcConfigurer {
4
5      @Override
6      public void
7          addViewControllers(ViewControllerRegistry
8              registry) {
9              registry
10             .addViewController("/")
11             .setViewName("home");
12         }
13     }

```

14.12 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
3      WebMvcConfigurer {

```

```

4     @Override
5     public void
6         configureViewResolvers(ViewResolverRegistry
7             registry) {
8             registry.enableContentNegotiation(new
9                 MappingJackson2JsonView());
10            registry.jsp();
11        }
12    }

```

14.13 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`.

14.14 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).

14.15 MVC Config API

In Java configuration, you can implement the WebMvcConfigurer interface:

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

@

15 REST Clients

The REST architectural style emphasises *uniform interfaces*, *independent deployment of components*, the *scalability of interactions* between them, and creating a layered architecture to promote *caching* to reduce user-perceived latency, enforce *security*, and encapsulate legacy systems.

- *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,  
5     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.

16 Integration Features

16.1 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 }
```

17 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.

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).

At Startup Time, Spring Boot Creates Spring MVC Components

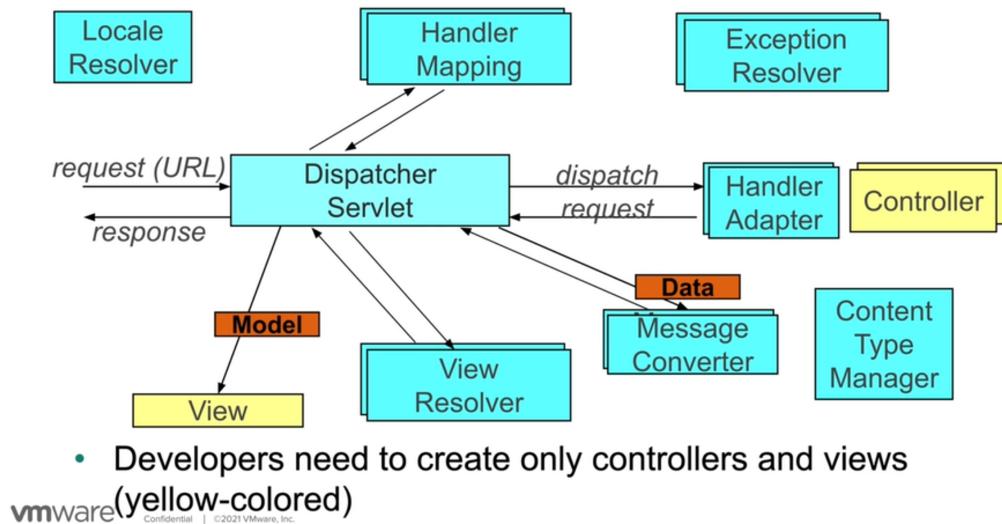


Figure 18: Spring Boot Web Architecture Overview

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

17.1 Spring Boot Starters

Main starter:

```
1 // spring-boot-starter/build.gradle
2
3 plugins {
4     id "org.springframework.boot.starter"
5 }
6
7 description = "Core starter, including
8     auto-configuration support, logging and YAML"
9
10 dependencies {
11     api(project(":spring-boot-project:spring-boot"))
12     api(project(":spring-boot-project:
13         spring-boot-autoconfigure"))
14     api(project(":spring-boot-project:
15         spring-boot-starters:spring-boot-starter-logging"))
```

```
15     api("jakarta.annotation:jakarta.annotation-api")
16     api("org.springframework:spring-core")
17     api("org.yaml:snakeyaml")
18 }
```

Examples:

```
1 // spring-boot-starter-test/build.gradle
2
3 plugins {
4     id "org.springframework.boot.starter"
5 }
6
7 description = "Starter for testing Spring Boot
8     applications with libraries including JUnit Jupiter,
9     Hamcrest and Mockito"
10
11 dependencies {
12     api(project(":spring-boot-project:spring-boot-starters:
13         spring-boot-starter"))
14     api(project(":spring-boot-project:spring-boot-test"))
15     api(project(":spring-boot-project:
16         spring-boot-test-autoconfigure"))
17     api("com.jayway.jsonpath:json-path")
18     api("jakarta.xml.bind:jakarta.xml.bind-api")
19     api("net.minidev:json-smart")
20     api("org.assertj:assertj-core")
21     api("org.awaitility:awaitility")
22     api("org.hamcrest:hamcrest")
23     api("org.junit.jupiter:junit-jupiter")
24     api("org.mockito:mockito-core")
25     api("org.mockito:mockito-junit-jupiter")
26     api("org.skyscreamer:jsonassert")
27     api("org.springframework:spring-core")
28     api("org.springframework:spring-test")
29     api("org.xmlunit:xmlunit-core") {
30         exclude group: "javax.xml.bind", module:
31             "jaxb-api"
32     }
33 }
34
35 checkRuntimeClasspathForConflicts {
36     ignore { name ->
37         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
8     Hibernate"
9
10 dependencies {
11     api(project(":spring-boot-project:spring-boot-starters:
12         spring-boot-starter"))
13     api(project(":spring-boot-project:spring-boot-starters:
14         spring-boot-starter-jdbc"))
15     api("org.hibernate.orm:hibernate-core")
16     api("org.springframework.data:spring-data-jpa")
17     api("org.springframework:spring-aspects")
18 }
```

```
1 // spring-boot-starter-aop/build.gradle
2
3 plugins {
4     id "org.springframework.boot.starter"
5 }
6
7 description = "Starter for aspect-oriented programming
8     with Spring AOP and AspectJ"
9
10 dependencies {
11     api(project(":spring-boot-project:spring-boot-starters:
12         spring-boot-starter"))
13     api("org.springframework:spring-aop")
14     api("org.aspectj:aspectjweaver")
15 }
```

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

Jackson is the preferred and default library.

Spring Boot contains the `spring-boot-devtools` dependency, enabling automatic application restart on file change.

17.2 Logging in Spring Boot

Spring Boot uses *Commons Logging* for all internal logging but leaves the underlying log implementation open.

Default configurations are provided for *Java Util Logging*, *Log4j2*, and *Logback*. In each case, loggers are pre-configured to use console output with optional file output also available.

By default, if you use the starters, *Logback* is used for logging. Appropriate Logback routing is also included to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J, or SLF4J all work correctly.

The default log level used by Spring Boot is INFO.

17.3 SpringApplication

Spring Boot's `SpringApplication` class is used to bootstrap and launch a Spring application from a Java main method. This class automatically creates the `ApplicationContext` from the classpath, scans the configuration classes and launches the application.

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.addLister` 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:

```
1 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)

17.4 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.

17.5 ApplicationContext in Boot

A `SpringApplication` attempts to create the right type of `ApplicationContext` on your behalf. The algorithm used to determine a `WebApplicationContext` 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(...)`.

17.6 Accessing Command Line Properties

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. Command line properties always take precedence over file-based property sources.

If you do not want command line properties to be added to the Environment, you can disable them by using `SpringApplication.setAddCommandLineProperties(false)`.

17.7 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).

17.8 Type-safe Configuration Properties in Spring Boot

Using the `@Value("${property}")` annotation to inject configuration properties can sometimes be cumbersome, especially if you are working with multiple properties or your data is hierarchical in nature. Spring Boot provides an alternative method of working with properties that lets strongly typed beans govern and validate the configuration of your application. TIP See also the differences between `@Value` and type-safe configuration properties.

17.8.1 JavaBean Properties Binding

It is possible to bind a bean declaring standard JavaBean properties as shown in the following example:

```
1 import java.net.InetAddress;
2 import java.util.ArrayList;
3 import java.util.Collections;
4 import java.util.List;
5 import
6         org.springframework.boot.context.properties.ConfigurationProperties
7
8 @ConfigurationProperties("my.service")
9 public class MyProperties {
10     private boolean enabled;
11     private InetAddress remoteAddress;
12     private final Security security = new Security();
13     public boolean isEnabled() {
14         return this.enabled;
15     }
16     public void setEnabled(boolean enabled) {
17         this.enabled = enabled;
18     }
19     public InetAddress getRemoteAddress() {
20         return this.remoteAddress;
21     }
22     public void setRemoteAddress(InetAddress
23         remoteAddress) {
24         this.remoteAddress = remoteAddress;
25     }
26     public Security getSecurity() {
27         return this.security;
28     }
29     public static class Security {
30         private String username;
31         private String password;
32         private List<String> roles = new
33             ArrayList<>(Collections.singleton("USER"));
34         public String getUsername() {
35             return this.username;
36         }
37         public void setUsername(String username) {
38             this.username = username;
39         }
40         public String getPassword() {
```

```

38         return this.password;
39     }
40     public void setPassword(String password) {
41         this.password = password;
42     }
43     public List<String> getRoles() {
44         return this.roles;
45     }
46     public void setRoles(List<String> roles) {
47         this.roles = roles;
48     }
49 }
50 }
```

The preceding POJO defines the following properties:

- my.service.enabled, with a value of false by default.
- my.service.remote-address, with a type that can be coerced from String.
- my.service.security.username, with a nested "security" object whose name is determined by the
- name of the property. In particular, the type is not used at all there and could have been
- SecurityProperties.
- my.service.security.password.
- my.service.security.roles, with a collection of String that defaults to USER.

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
2     org.springframework.context.annotation.Configuration;
3 import org.springframework.context.annotation.Profile;
4
5 @Configuration(proxyBeanMethods = false)
6 @Profile("production")
7 public class ProductionConfiguration {
8     // ...
9 }
```

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.

17.8.2 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:

```
1 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:

```
1 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:

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

17.8.3 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:

```
1 spring.profiles.include[0]=common  
2 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.

17.8.4 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`.

17.8.5 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.

17.8.6 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`

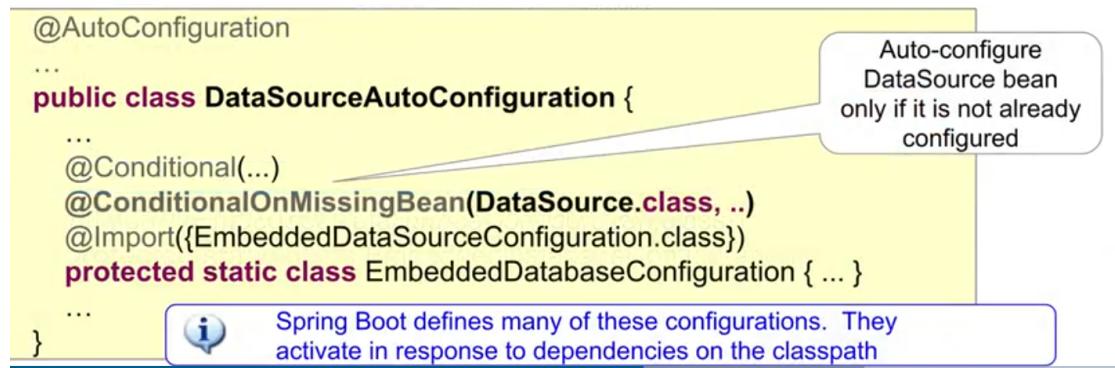


Figure 19: Spring’s DataSourceAutoConfiguration class.

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.

17.9 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.AutoConfigure` lists the classes by default autoconfigured by Spring.

Spring’s `DataSourceAutoConfiguration` class is one example. See fig. 19.

17.10 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.
- 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.
- AAwaitility: A library for testing asynchronous systems.

17.11 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
4      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.

17.12 Test Annotations

17.12.1 `@SpringJUnitConfig`

`@SpringJUnitConfig` is a composed annotation that combines `@ExtendWith(SpringExtension.class)` from JUnit Jupiter with `@ContextConfiguration` from the Spring TestContext Framework.

It can be used at the class level as a drop-in replacement for `@ContextConfiguration`. With regard to configuration options, the only difference between `@ContextConfiguration` and `@SpringJUnitConfig` is that component classes may be declared with the `value` attribute in `@SpringJUnitConfig`.

The following example shows how to use the `@SpringJUnitConfig` annotation to specify a configuration class:

```
1  @SpringJUnitConfig(TestConfig.class)
2  class ConfigurationClassJUnitJupiterSpringTests {
3      // class body...
4  }
```

The following example shows how to use the `@SpringJUnitConfig` annotation to specify the location of a configuration file:

```
1  @SpringJUnitConfig(locations = "/test-config.xml")
2  class XmlJUnitJupiterSpringTests {
3      // class body...
4  }
```

17.12.2 `@SpringBootTest`

The `@SpringBootTest` annotation is used to specify the main class that should be used when launching an application context for integration tests.

By default, `@SpringBootTest` does not start a 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
7          Exception {
8          mvc.perform(get("/")).andExpect(status().isOk()).andExpect(content().string("Hello
9          World"));
10     }
11
12     // If Spring WebFlux is on the classpath, you can
13     // drive MVC tests with a WebTestClient
14     @Test
```

```

12     void testWithWebTestClient(@Autowired WebTestClient
13         webClient) {
14     webClient
15         .get().uri("/")
16         .exchange()
17         .expectStatus().isOk()
18         .expectBody(String.class).isEqualTo("Hello
19             World");
20     }

```

SpringBootTest is itself annotated with `@ExtendWith(org.springframework.test.context.junit.jupiter.SpringTestContextExtension.class)`. It provides the following features over and above the regular Spring TestContext Framework:

- Uses `SpringBootTestLoader` as the default ContextLoader when no specific `@ContextConfiguration(loader=...)` is defined.
- Automatically searches for a `@SpringBootConfiguration` when nested `@Configuration` is not used, and no explicit classes are specified.
- Allows *custom Environment properties* to be defined using the `properties` attribute.
- Allows application arguments to be defined using the `args` attribute.
- Provides support for different `webEnvironment` modes, including the ability to start a fully running web server listening on a defined or random port.
- Registers a `TestRestTemplate` and/or `WebTestClient` bean for use in web tests that are using a fully running web server.

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 `webApplicationContext` 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. Alternatively, the property `server.port` may be set to 0. The `@LocalServerPort` annotation - equivalent to `@Value("${local.server.port}")` - can be used to inject the *actual port used* into your test.

- 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).

17.12.3 @WebMvcTest

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 - but not @Component, @Service or @Repository beans.

Auto-configured capabilities include: a templating engine (Thymeleaf), Jackson, caching.

By default, tests annotated with @WebMvcTest will also auto-configure Spring Security and MockMvc (include support for HtmlUnit WebClient and Selenium WebDriver). For more fine-grained control of MockMVC the @AutoConfigureMockMvc annotation can be used.

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.

17.12.4 Auto-Configuration for Slice Tests

Auto-configured JPA Tests The annotation @DataJpaTest causes the following auto-configuration to take place:

```

1 // optional: org.springframework.boot.testcontainers.
2 service.connection.ServiceConnectionAutoConfiguration
3
4 // caching
5 // org.springframework.boot.autoconfigure.
6 cache.CacheAutoConfiguration
7
8 // repositories
9 data.jpa.JpaRepositoriesAutoConfiguration
10
11 // data sources and jdbc access
12 flyway.FlywayAutoConfiguration

```

```

13    jdbc.DataSourceAutoConfiguration
14      jdbc.DataSourceTransactionManagerAutoConfiguration
15        jdbc.JdbcClientAutoConfiguration
16          jdbc.JdbcTemplateAutoConfiguration
17            liquibase.LiquibaseAutoConfiguration
18
19 // testing setup
20 //org.springframework.boot.test.
21 autoconfigure.jdbc.TestDatabaseAutoConfiguration
22     autoconfigure.orm.jpa.TestEntityManagerAutoConfiguration

```

Note especially the auto-configuration of caching, JDBCTemplate, Liquibase and Flyway.

17.12.5 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 test GetUserById() {
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`.

17.13 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
8     org.assertj.core.api.Assertions.assertThat;
9
10 class MyTests {
11
12     private final TestRestTemplate template = new
13         TestRestTemplate();
14
15     @Test
16     void testRequest() {
17         ResponseEntity<String> headers =
18             this.template.getForEntity(
19                 "https://myhost.example.com/example",
20                 String.class);
21         assertThat(
22             headers.getHeaders().getLocation()).hasHost(
23                 "other.example.com");
24     }
25 }
```

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
6     org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.boot.test.context.SpringBootTest;
8 import org.springframework.boot.test.context.SpringBootTest.
9     WebEnvironment;
9 import
10    org.springframework.boot.test.context.TestConfiguration;
10 import
11    org.springframework.boot.test.web.client.TestRestTemplate;
11 import
12    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 =
26             this.template.getForEntity("/example",
27                 String.class).getHeaders();
28         assertThat(headers.getLocation())
29             .hasHost("other.example.com");
30     }
31
32     @TestConfiguration(proxyBeanMethods = false)
33     static class RestTemplateBuilderConfiguration {
```

```

33     @Bean
34     RestTemplateBuilder restTemplateBuilder() {
35         return new
36             RestTemplateBuilder().setConnectTimeout(
37                 Duration.ofSeconds(1))
38             .setReadTimeout(Duration.ofSeconds(1));
39     }
40 }
41 }
42 }
```

17.14 Spring Boot Actuator

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

17.14.1 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.

The following technology-agnostic endpoints are available:

- auditevents
- beans
- caches
- conditions
- configprops
- env
- flyway
- health
- httpexchanges
- info
- integrationgraph
- loggers

- liquibase
- metrics
- mappings
- quartz
- scheduledtasks
- sessions
- shutdown
- startup
- threaddump

If your application is a web application (Spring MVC, Spring WebFlux, or Jersey), you can use the following additional endpoints:

- heapdump
- logfile
- prometheus

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.enabled-by-default 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:

```
1 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:

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

Monitoring and Management Over HTTP Customization:

```
1 // base for all endpoints
2 management.endpoints.web.base-path=
3
4 // for a specific endpoint
5 management.endpoints.web.path-mapping.<actuator>
6 // e.g.
7 management.endpoints.web.path-mapping.health=custom-health
8
9 // port
10 management.server.port=8081
```

Health information By default, Spring Boot defines four different values for health status:

- UP — The component or subsystem is working as expected

- DOWN — The component is not working
- OUT_OF_SERVICE — The component is out of service temporarily
- UNKNOWN — The component state is unknown

Information exposed by the health endpoint depends on the `management.endpoint.health.show-detail` 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. Auto-configured are:

```

1  cassandra
2  couchbase
3  db
4  diskspace
5  elasticsearch
6  hazelcast
7  influxdb
8  jms
9  ldap
10 mail
11 mongo
12 neo4j
13 ping
14 rabbit
15 redis

```

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.

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:

```

1 import org.springframework.boot.actuate.health.Health;
2 import
3     org.springframework.boot.actuate.health.HealthIndicator;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class MyHealthIndicator implements
8     HealthIndicator {
9
10    @Override
11    public Health health() {
12        int errorCode = check();
13        if (errorCode != 0) {
14            return Health.down().withDetail("Error
15                Code", errorCode).build();
16        }
17        return Health.up().build();
18    }
19
20    private int check() {
21        // perform some specific health check
22        return ...
23    }
24 }
```

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
4 // name starts with info
5 env
6 // git.properties resource exists
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:

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

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

```
1 import java.util.Collections;
2
3 import org.springframework.boot.actuate.info.Info;
4 import
5     org.springframework.boot.actuate.info.InfoContributor;
6 import org.springframework.stereotype.Component;
7
8 @Component
9 public class MyInfoContributor implements
10    InfoContributor {
11
12    @Override
13    public void contribute(Info.Builder builder) {
14        builder.withDetail("example",
15            Collections.singletonMap("key", "value"));
16    }
17}
```

14

15 }

Metrics Areas where metrics are recorded include:

- JVM Metrics
- System Metrics
- Application Startup Metrics
- Logger Metrics
- Task Execution and Scheduling Metrics
- JMS Metrics
- Spring MVC Metrics

as well as product-specific metrics.

Tracing Spring Boot Actuator provides dependency management and auto-configuration for Micrometer Tracing, a facade for popular tracer libraries.

Spring Boot ships auto-configuration for the following tracers:

- OpenTelemetry with Zipkin, Wavefront, or OTLP
- OpenZipkin Brave with Zipkin or Wavefront

Recording HTTP Exchanges You can enable recording of HTTP exchanges by providing a bean of type `HttpExchangeRepository` in your application's configuration. For convenience, Spring Boot offers `InMemoryHttpExchangeRepository`, which, by default, stores the last 100 request-response exchanges.

You can use the `httpexchanges` endpoint to obtain information about the request-response exchanges that are stored in the `HttpExchangeRepository`.

To customize the items that are included in each recorded exchange, use the `management.httpexchanges` configuration property.

To disable recording entirely, set `management.httpexchanges.recording.enabled` to false.

Auditing Once Spring Security is in play, Spring Boot Actuator has a flexible audit framework that publishes events (by default, “authentication success”, “failure” and “access denied” exceptions). This feature can be very useful for reporting and for implementing a lock-out policy based on authentication failures.

You can enable auditing by providing a bean of type `AuditEventRepository` in your application’s configuration. For convenience, Spring Boot offers an `InMemoryAuditEventRepository`.

To customize published security events, you can provide your own implementations of `AbstractAuthenticationAuditListener` and `AbstractAuthorizationAuditListener`.

You can also use the audit services for your own business events. To do so, either inject the `AuditEventRepository` bean into your own components and use that directly or publish an `AuditApplicationEvent` with the Spring `ApplicationEventPublisher` (by implementing `ApplicationEventPublisherAware`).

18 Special features

18.1 HATEOAS

If you develop a RESTful API that makes use of hypermedia, Spring Boot provides auto-configuration for Spring HATEOAS that works well with most applications.

The auto-configuration replaces the need to use `@EnableHypermediaSupport` and registers a number of beans to ease building hypermedia-based applications, including a `LinkDiscoverer` (for client side support) and an `ObjectMapper` configured to correctly marshal responses into the desired representation.

The `ObjectMapper` is customized by setting the various `spring.jackson.*` properties or, if one exists, by a `Jackson2ObjectMapperBuilder` bean.

Note: You can take control of Spring HATEOAS’s configuration by explicitly using `@EnableHypermediaSupport`. Doing so *disables* the `ObjectMapper` customization described earlier.