Spring Bean Lifecycle

Providing an Inversion-of-Control Container is one of the core provisions of the Spring Framework. Spring orchestrates the beans in its application context and manages their lifecycle. In this tutorial, we're looking at the lifecycle of those beans and how we can hook into it.

What Is a Spring Bean?

Let's start with the basics. Every object that is under the control of Spring's `ApplicationContext` in terms of *creation*, *orchestration*, and *destruction* is called a Spring Bean.

The most common way to define a Spring bean is using the `@Component` annotation:

```
@Component
class MySpringBean {
  ...
}
```

If Spring's component scanning is enabled, an object of `MySpringBean` will be added to the application context.

Another way is using Spring's Java config:

```
@Configuration
class MySpringConfiguration {

  @Bean
  public MySpringBean mySpringBean() {
    return new MySpringBean();
  }

}
```

The Spring Bean Lifecycle

When we look into the lifecycle of Spring beans, we can see numerous phases starting from the object instantiation up to their destruction.

Let's explain these phases in a little bit more detail.

Bean Creation Phases

Instantiation: This is where everything starts for a bean. Spring instantiates bean objects just like we would manually create a Java object instance.

Populating Properties: After instantiating objects, Spring scans the beans that implement `Aware` interfaces and starts setting relevant properties.

Pre-Initialization: Spring's `BeanPostProcessor`s get into action in this phase. The `postProcessBeforeInitialization()` methods do their job. Also, `@PostConstruct` annotated methods run right after them.

AfterPropertiesSet: Spring executes the `afterPropertiesSet()` methods of the beans which implement `InitializingBean`.

Custom Initialization: Spring triggers the initialization methods that we defined in the `initMethod` attribute of our `@Bean`annotations.

Post-Initialization: Spring's `BeanPostProcessor`s are in action for the second time. This phase triggers the `postProcessAfterInitialization()` methods.

Bean Destruction Phases

Pre-Destroy: Spring triggers`@PreDestroy` annotated methods in this phase.

Destroy: Spring executes the `destroy()` methods of `DisposableBean` implementations.

Custom Destruction: We can define custom destruction hooks with the `destroyMethod` attribute in the `@Bean` annotation and Spring runs them in the last phase.

Hooking Into the Bean Lifecycle

There are numerous ways to hook into the phases of the bean lifecycle in a Spring application.

Let's see some examples for each of them.

Using Spring's Interfaces

We can implement Spring's `InitializingBean` interface to run custom operations in `afterPropertiesSet()` phase:

```
@Component
class MySpringBean implements InitializingBean {

  @Override
  public void afterPropertiesSet() {
    //...
  }

}
```

Similarly, we can implement `DisposableBean` to have Spring call the `destroy()` method in the destroy phase:

```
@Component
class MySpringBean implements DisposableBean {

  @Override
  public void destroy() {
    //...
  }

}
```

Using JSR-250 Annotations

Spring supports the `@PostConstruct` and `@PreDestroy` annotations of the JSR-250 specification.

Therefore, we can use them to hook into the pre-initialization and destroy phases:

```
@Component
class MySpringBean {

  @PostConstruct
  public void postConstruct() {
    //...
  }

  @PreDestroy
  public void preDestroy() {
    //...
  }

}
```

Using Attributes of the `@Bean` Annotation

Additionally, when we define our Spring beans we can set the `initMethod` and `destroyMethod` attributes of the `@Bean` annotation in Java configuration:

```
@Configuration
class MySpringConfiguration {
```

```
  @Bean(initMethod = "onInitialize", destroyMethod = "onDestroy")
  public MySpringBean mySpringBean() {
    return new MySpringBean();
  }

}
```

We should note that **if we have a public method named `close()` or `shutdown()` in our bean, then it is automatically triggered with a destruction callback by default**:

```
@Component
class MySpringBean {

  public void close() {
    //...
  }

}
```

However, if we do not wish this behavior, we can disable it by setting `destroyMethod=""`:

```
@Configuration
class MySpringConfiguration {

  @Bean(destroyMethod = "")
  public MySpringBean mySpringBean() {
    return new MySpringBean();
  }

}
```

XML Configuration

For legacy applications, we might have still some beans left in XML configuration. Luckily, we can still configure these attributes in our XML bean definitions.

Using `BeanPostProcessor`

Alternatively, we can make use of the `BeanPostProcessor` interface to be able to run any custom operation before or after a Spring bean initializes and even return a modified bean:

```
class MyBeanPostProcessor implements BeanPostProcessor {

  @Override
  public Object postProcessBeforeInitialization(Object bean, String beanName)
    throws BeansException {
    //...
    return bean;
  }

  @Override
  public Object postProcessAfterInitialization(Object bean, String beanName)
    throws BeansException {
    //...
    return bean;
  }

}
```

`BeanPostProcessor` Is Not Bean Specific

We should pay attention that Spring's `BeanPostProcessor`s are executed for each bean defined in the spring context.

Using `Aware` Interfaces

Another way of getting into the lifecycle is by using the `Aware` interfaces:

```
@Component
class MySpringBean implements BeanNameAware, ApplicationContextAware {

  @Override
  public void setBeanName(String name) {
    //...
  }

  @Override
  public void setApplicationContext(ApplicationContext applicationContext)
    throws BeansException {
    //...
  }

}
```

There are additional `Aware` interfaces which we can use to inject certain aspects of the Spring context into our beans.

Why Would I Need to Hook Into the Bean Lifecycle?

When we need to extend our software with new requirements, it is critical to find the best practices to keep our codebase maintainable in the long run.

In Spring Framework, **hooking into the bean lifecycle is a good way to extend our application in most cases.**

Acquiring Bean Properties

One of the use cases is acquiring the bean properties (like bean name) at runtime. For example, when we do some logging:

```
@Component
class NamedSpringBean implements BeanNameAware {

  Logger logger = LoggerFactory.getLogger(NamedSpringBean.class);

  public void setBeanName(String name) {
    logger.info(name + " created.");
  }

}
```

Dynamically Changing Spring Bean Instances

In some cases, we need to define Spring beans programmatically. This can be a practical solution when we need to re-create and change our bean instances at runtime.

Let's create an `IpToLocationService` service which is capable of dynamically updating `IpDatabaseRepository` to the latest version on-demand:

```
@Service
class IpToLocationService implements BeanFactoryAware {

  DefaultListableBeanFactory listableBeanFactory;
  IpDatabaseRepository ipDatabaseRepository;

  @Override
  public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
    listableBeanFactory = (DefaultListableBeanFactory) beanFactory;
    updateIpDatabase();
  }

  public void updateIpDatabase(){
    String updateUrl = "https://download.acme.com/ip-database-latest.mdb";

    AbstractBeanDefinition definition = BeanDefinitionBuilder
        .genericBeanDefinition(IpDatabaseRepository.class)
```

```
            .addPropertyValue("file", updateUrl)
            .getBeanDefinition();

        listableBeanFactory
            .registerBeanDefinition("ipDatabaseRepository", definition);

        ipDatabaseRepository = listableBeanFactory
            .getBean(IpDatabaseRepository.class);
    }
}
```

We access the `BeanFactory` instance with the help of `BeanFactoryAware` interface. Thus, we dynamically create our `IpDatabaseRepository` bean with the latest database file and update our bean definition by registering it to the Spring context.

Also, we call our `updateIpDatabase()` method right after we acquire the `BeanFactory` instance in the `setBeanFactory()` method. Therefore, we can initially create the first instance of the `IpDatabaseRepository` bean while the Spring context boots up.

Accessing Beans From the Outside of the Spring Context

Another scenario is accessing the `ApplicationContext` or `BeanFactory` instance from outside of the Spring context.

For example, we may want to inject the `BeanFactory` into a non-Spring class to be able to access Spring beans or configurations inside that class. The integration between Spring and the Quartz library is a good example to show this use case:

```
class AutowireCapableJobFactory
    extends SpringBeanJobFactory implements ApplicationContextAware {

  private AutowireCapableBeanFactory beanFactory;

  @Override
  public void setApplicationContext(final ApplicationContext context) {
    beanFactory = context.getAutowireCapableBeanFactory();
  }

  @Override
  protected Object createJobInstance(final TriggerFiredBundle bundle)
      throws Exception {
    final Object job = super.createJobInstance(bundle);
    beanFactory.autowireBean(job);
    return job;
  }

}
```

In this example, we're using the `ApplicationContextAware` interface to get access to the bean factory and use the bean factory to autowire the dependencies in a `Job` bean that is initially not managed by Spring.

Also, a common Spring - Jersey integration is another clear example of this:

```
@Configuration
class JerseyConfig extends ResourceConfig {

  @Autowired
  private ApplicationContext applicationContext;

  @PostConstruct
  public void registerResources() {
    applicationContext.getBeansWithAnnotation(Path.class).values()
      .forEach(this::register);
  }

}
```

By marking Jersey's `ResourceConfig` as a Spring `@Configuration`, we inject the `ApplicationContext` and lookup all the beans which are annotated by Jersey's `@Path`, to easily register them on application startup.

The Execution Order

Let's write a Spring bean to see the execution order of each phase of the lifecycle:

```
class MySpringBean implements BeanNameAware, ApplicationContextAware,
    InitializingBean, DisposableBean {

  private String message;

  public void sendMessage(String message) {
    this.message = message;
  }

  public String getMessage() {
    return this.message;
  }

  @Override
  public void setBeanName(String name) {
    System.out.println("--- setBeanName executed ---");
  }

  @Override
  public void setApplicationContext(ApplicationContext applicationContext)
      throws BeansException {
    System.out.println("--- setApplicationContext executed ---");
  }

  @PostConstruct
  public void postConstruct() {
    System.out.println("--- @PostConstruct executed ---");
  }

  @Override
  public void afterPropertiesSet() {
    System.out.println("--- afterPropertiesSet executed ---");
  }

  public void initMethod() {
    System.out.println("--- init-method executed ---");
  }

  @PreDestroy
  public void preDestroy() {
    System.out.println("--- @PreDestroy executed ---");
  }

  @Override
  public void destroy() throws Exception {
    System.out.println("--- destroy executed ---");
  }

  public void destroyMethod() {
    System.out.println("--- destroy-method executed ---");
  }

}
```

Additionally, we create a `BeanPostProcessor` to hook into the before and after initialization phases:

```
class MyBeanPostProcessor implements BeanPostProcessor {

  @Override
  public Object postProcessBeforeInitialization(Object bean, String beanName)
      throws BeansException {
    if (bean instanceof MySpringBean) {
      System.out.println("--- postProcessBeforeInitialization executed ---");
    }
    return bean;
  }

  @Override
  public Object postProcessAfterInitialization(Object bean, String beanName)
      throws BeansException {
    if (bean instanceof MySpringBean) {
      System.out.println("--- postProcessAfterInitialization executed ---");
```

```
    }
    return bean;
  }

}
```

Next, we write a Spring configuration to define our beans:

```
@Configuration
class MySpringConfiguration {

  @Bean
  public MyBeanPostProcessor myBeanPostProcessor(){
    return new MyBeanPostProcessor();
  }

  @Bean(initMethod = "initMethod", destroyMethod = "destroyMethod")
  public MySpringBean mySpringBean(){
    return new MySpringBean();
  }

}
```

Finally, we write a `@SpringBootTest` to run our Spring context:

```
@SpringBootTest
class BeanLifecycleApplicationTests {

  @Autowired
  public MySpringBean mySpringBean;

  @Test
  public void testMySpringBeanLifecycle() {
    String message = "Hello World";
    mySpringBean.sendMessage(message);
    assertThat(mySpringBean.getMessage()).isEqualTo(message);
  }

}
```

As a result, our test method logs the execution order between the lifecycle phases:

```
--- setBeanName executed ---
--- setApplicationContext executed ---
--- postProcessBeforeInitialization executed ---
--- @PostConstruct executed ---
--- afterPropertiesSet executed ---
--- init-method executed ---
--- postProcessAfterInitialization executed ---
...
--- @PreDestroy executed ---
--- destroy executed ---
--- destroy-method executed ---
```

Conclusion

In this tutorial, we learned what the bean lifecycle phases are, why, and how we hook into lifecycle phases in Spring.

Spring has numerous phases in a bean lifecycle as well as many ways to receive callbacks. We can hook into these phases both via annotations on our beans or from a common class as we do in `BeanPostProcessor`.

Although each method has its purpose, **we should note that using Spring interfaces couples our code to the Spring Framework.**

On the other hand, **`@PostConstruct` and `@PreDestroy` annotations are a part of the Java API. Therefore, we consider them a better alternative to receiving lifecycle callbacks because they decouple our components even from Spring.**