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

TBD https://www.baeldung.com/spring-annotations-resource-inject-autowire

2 Dependency injection

2.1 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.

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

Resp.

```
1        <bean id="item1" class="org.baeldung.store.ItemImpl1"
             />
2        <bean id="store" class="org.baeldung.store.Store">
```

```
3          <constructor-arg type="ItemImpl1" index="0"
               name="item" ref="item1" />
4          </bean>
```

2.2 Setter-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>
```

2.3 Field-based

In case of Field-Based DI, we can inject the dependencies by marking them with an @Autowired annotation:

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

Drawbacks:

- This method uses reflection to inject the dependencies, which is costlier than constructor-based or setter-based injection.

- It's easy to keep adding multiple dependencies using this approach. If we were using constructor injection, having multiple arguments would make us think that the class does more than one thing, which can violate the Single Responsibility Principle.

3 Annotations

3.1 Annotations for dependency injection

3.1.1 @Autowired

@Autowired marks a dependency which Spring is going to resolve and inject. We can use this annotation with a 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.

3.1.2 @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.

3.1.3 @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;
3        }
```

3.1.4 @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          }
```

3.1.5 @Lazy

This annotation behaves differently depending on where we exactly 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         }
```

3.1.6 @Scope

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

3.2 Context Configuration Annotations

3.2.1 @Profile

If we want Spring to use a @Component class or a @Bean method only when a specific profile is active, we can mark it with @Profile.

```
1          @Component
2          @Profile("sportsDay")
3          class Bike implements Vehicle {}
```

3.3 Context Configuration Annotations

3.3.1 @Import

We can use specific @Configuration classes without component scanning with this annotation. We can provide those classes with @Import's value argument.

```
1          @Import(VehiclePartSupplier.class)
2          class VehicleFactoryConfig {}
```

3.3.2 @ImportResource

We can import XML configurations with this annotation. 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 {}
```

3.3.3 @PropertySource

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

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

3.4 Bean annotations

Three ways to configure beans:

- XML configuration

- Annotation-based, using @Bean

- Component scanning, using annotations from org.springframework.stereotype

3.4.1 @ComponentScan

@ComponentScan configures which packages to scan for classes with annotation configuration. E.g., specifying packages:

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

Or else, specifying classes:

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

The @ComponentScan annotation is used together with @Configuration.

3.5 @ComponentScan Without Arguments

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

3.5.1 @ComponentScan With Arguments

Passing basePackages, we can, for example, change the base package.

```
1          @ComponentScan ( basePackages =
               "com.baeldung.componentscan.springapp.animals")
2          @Configuration
3          public class SpringComponentScanApp {
4              // ...
5          }
```

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

```
1          @ComponentScan ( basePackages =
               "com.baeldung.componentscan.springapp.animals;com.baeldung.compone
2          @ComponentScan ( basePackages =
               "com.baeldung.componentscan.springapp.animals ,com.baeldung.compone
```

```
3          @ComponentScan(basePackages =
               "com.baeldung.componentscan.springapp.animals
               com.baeldung.componentscan.springapp.flowers")
```

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

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

Or:

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

3.5.2 @Component

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

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

@Repository, @Service, @Configuration, and @Controller are all meta-annotations of @Component. Spring also automatically picks them up during the component scanning process.

3.5.3 @Repository

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

3.5.4 @Service

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

3.5.5 @Controller

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

3.5.6 @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        }
```

3.6 Spring Boot Annotations

3.6.1 @SpringBootApplication

This is a combination of three annotations:

```
1        @Configuration
2        @EnableAutoConfiguration
3        @ComponentScan
```

3.7 Comparing @Resource, @Autowired, and @Inject annotations

3.7.1 @Resource

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

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

3.7.2 @Inject

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 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 MyAppContext {
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=AnnotationConfigContextLoader.class,
```

```
12            classes= MyAppContext.class)
13        public class Xxx {
14            @Inject
15            private File defaultFile;
16        }
```

3.8 @Autowired

TBD

4 Aware Interfaces

Indicates that the bean is eligible to be notified by the Spring container through the callback methods. The 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.

4.1 BeanNameAware

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

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

4.2 BeanFactoryAware

Provides access to the BeanFactory which created the object.

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

4.3 ApplicationContextAware

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

5 Bean Lifecycle

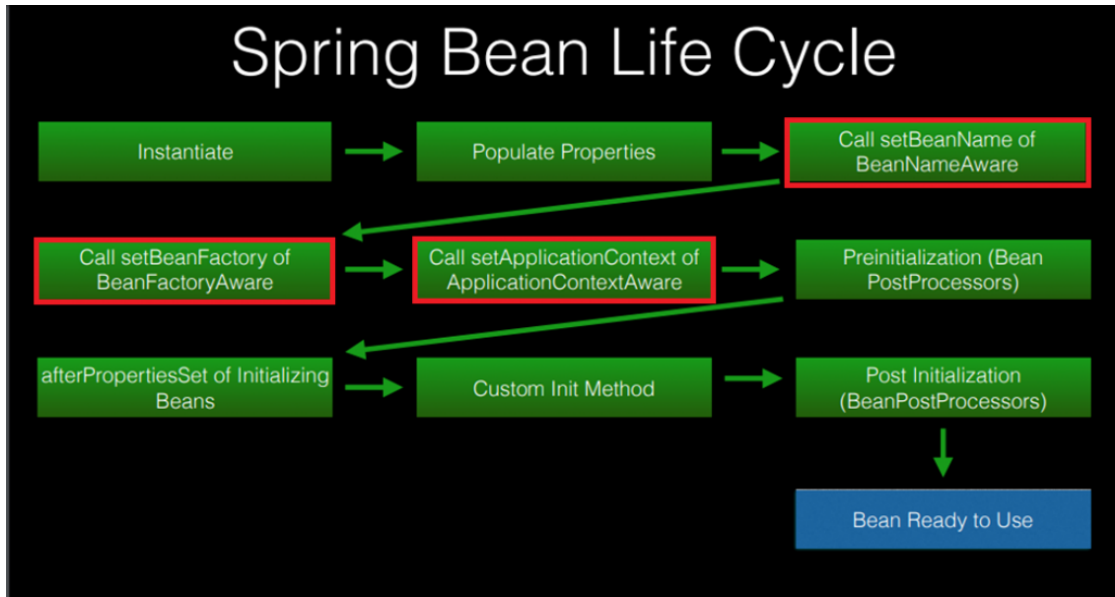## 5.1 Overview
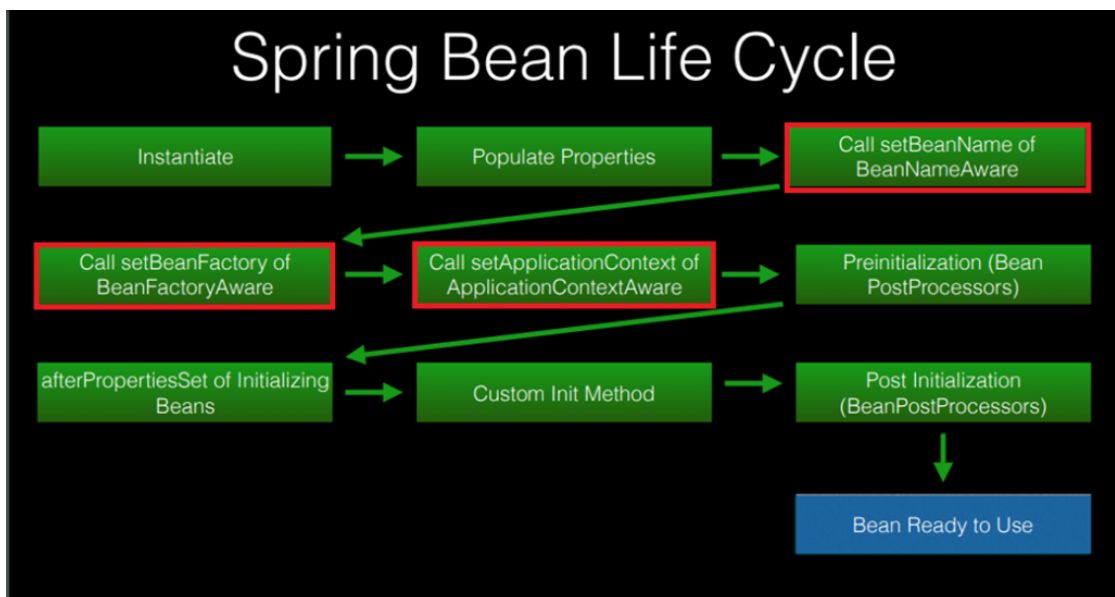
Zooming in on the pre-instantation changes:
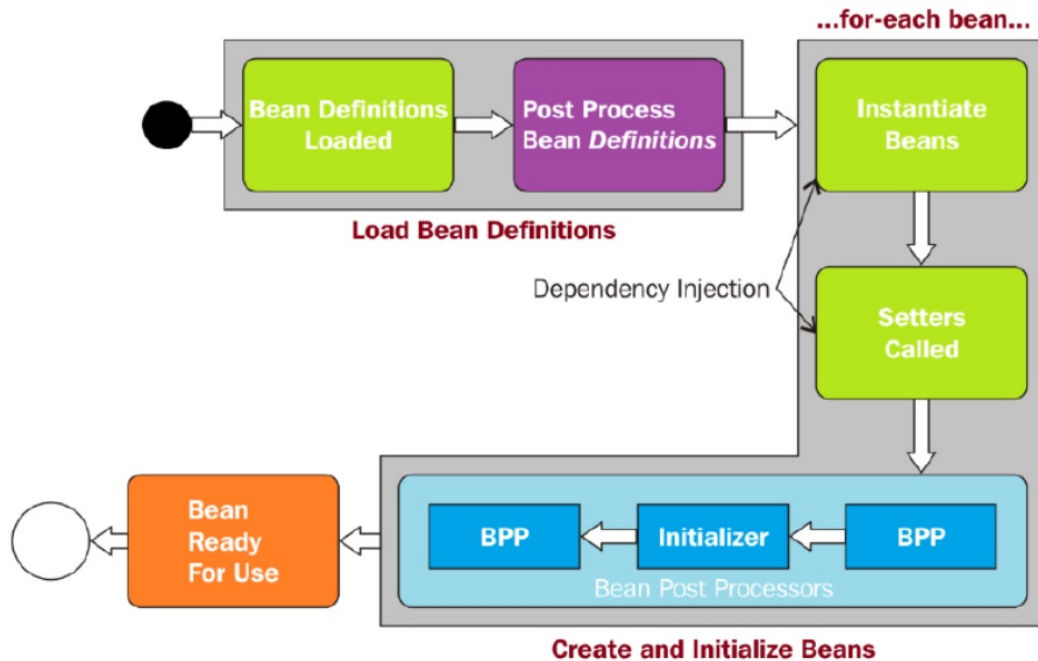
Figure 1:

Figure 2:

Figure 3:

See https://www.digitalocean.com/community/tutorials/spring-bean-life-cycle for code to display the order of invocations.

5.1.1 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.

5.1.2 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 your 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.

5.1.3 Instantiate beans

Injects values and bean references into beans' properties.

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

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

5.1.6 Call ApplicationContextAware's setApplicationContext for each bean implementing it

5.1.7 Run pre-initialization BeanPostProcessors

The Application context calls postProcessBeforeInitialization() for each bean implementing BeanPostProcessor. A bean implementing BeanFactoryPostProcessor is called when all bean definitions have been loaded, but no beans will have been instantiated yet. This allows for overriding or adding properties even to eager-initializing beans. This will let you have access to all the beans that you have defined in XML or that are annotated (scanned via component-scan).

5.1.8 Call InitializingBean's afterPropertiesSet()

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

5.1.9 Init Method

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

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

And here, the @PostConstruct annotation.

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

5.1.10 Run post-initialization BeanPostProcessors

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

5.1.11 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.

5.1.12 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.

6 Bean Naming

6.1 Default Bean Naming

6.1.1 Class-level

For an annotation used at the class, Spring uses the class name and converts the first letter to lowercase. The same default naming strategy is applicable for all class-level annotations that are used to create a Spring bean, such as @Component, @Service, and @Controller.

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

6.1.2 Method-level

When we use the @Bean annotation on a method, Spring uses the method name as a 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          }
```

Similar to @Component("myBean"), we can specify the name using other annotations such as @Service("myService"), @Controller("myController"), and @Bean("myCustomBean").

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,
                  @Qualifier("cat") Animal cat) {
23              this.dog = dog;
24              this.cat = cat;
25          }
26          public Animal getDog() {
27              return dog;
28          }
29          public Animal getCat() {
30              return cat;
31          }
32      }
```