Bean Factory and Dependency Injection

jim stafford

Fall 2024 2023-06-25: Built: 2024-11-19 21:30 EST

Table of Contents

1. Introduction . . . 1
  1.1. Goals . . . 1
  1.2. Objectives . . . 1
2. Hello Service . . . 2
  2.1. Hello Service API . . . 2
  2.2. Hello Service StdOut . . . 2
  2.3. Hello Service API pom.xml . . . 3
  2.4. Hello Service StdOut pom.xml . . . 3
  2.5. Hello Service Interface . . . 4
  2.6. Hello Service Sample Implementation . . . 4
  2.7. Hello Service Modules Complete . . . 5
  2.8. Hello Service API Maven Build . . . 5
  2.9. Hello Service StdOut Maven Build . . . 6
3. Application Module . . . 8
  3.1. Application Maven Dependency . . . 8
  3.2. Viewing Dependencies . . . 9
  3.3. Application Java Dependency . . . 9
4. Dependency Injection . . . 11
5. Spring Dependency Injection . . . 12
  5.1. @Autowired Annotation . . . 12
  5.2. Dependency Injection Flow . . . 13
6. Bean Missing . . . 14
  6.1. Bean Missing Error Solution(s) . . . 14
7. @Configuration classes . . . 15
8. @Bean Factory Method . . . 16
9. @Bean Factory Used . . . 17
10. Factory Alternative: XML Configuration . . . 18
11. @Configuration Alternatives . . . 20
  11.1. Example Lifecycle POJO . . . 20
  11.2. Example Lifecycle @Configuration Class . . . 21
  11.3. Consuming @Bean Factories within Same Class . . . 21
  11.4. Using Defaults (Singleton, Proxy) . . . 21
  11.5. Prototype Scope, Proxy True . . . 22
  11.6. Prototype Scope, Proxy False . . . 23
  11.7. Singleton Scope, Proxy False . . . 23
  11.8. @Configuration Takeaways . . . 24
12. Summary . . . 25

Chapter 1. Introduction

This material provides an introduction to configuring an application using a factory method. This is the most basic use of separation between the interface used by the application and the decision of what the implementation will be.

The configuration choice shown will be part of the application but as you will see later, configurations can be deeply nested — far away from the details known to the application writer.

1.1. Goals

The student will learn:

- to decouple an application through the separation of interface and implementation
- to configure an application using dependency injection and factory methods of a configuration class

1.2. Objectives

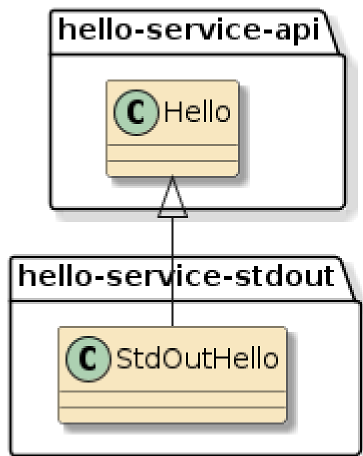
At the conclusion of this lecture and related exercises, the student will be able to:

1. implement a service interface and implementation component
2. package a service within a Maven module separate from the application module
3. implement a Maven module dependency to make the component class available to the application module
4. use a `@Bean` factory method of a `@Configuration` class to instantiate a Spring-managed component

Chapter 2. Hello Service

To get started, we are going to create a simple `Hello` service. We are going to implement an interface and a single implementation right off the bat. They will be housed in two separate modules:

- hello-service-api
- hello-service-stdout



We will start out by creating two separate module directories.

2.1. Hello Service API

The Hello Service API module will contain a single interface and pom.xml.

```
hello-service-api/
|-- pom.xml
`-- src
    `-- main
        `-- java
            `-- info
                `-- ejava
                    `-- examples
                        `-- app
                            `-- hello
                                `-- Hello.java (1)
```

(1) Service interface

2.2. Hello Service StdOut

The Hello Service StdOut module will contain a single implementation class and pom.xml.

```
hello-service-stdout/
|-- pom.xml
`-- src
```

```
    `-- main
        `-- java
            `-- info
                `-- ejava
                    `-- examples
                        `-- app
                            `-- hello
                                `-- stdout
                                    `-- StdOutHello.java ①
```

① Service implementation

2.3. Hello Service API pom.xml

We will be building a normal Java JAR with no direct dependencies on Spring Boot or Spring.

hello-service-api pom.xml

```
#pom.xml
...
    <groupId>info.ejava.examples.app</groupId>
    <version>6.1.0-SNAPSHOT</version>
    <artifactId>hello-service-api</artifactId>
    <packaging>jar</packaging>
...
```

2.4. Hello Service StdOut pom.xml

The implementation will be similar to the interface's pom.xml except it requires a dependency on the interface module.

hello-service-stdout pom.xml

```
#pom.xml
...
    <groupId>info.ejava.examples.app</groupId>
    <version>6.1.0-SNAPSHOT</version>
    <artifactId>hello-service-stdout</artifactId>
    <packaging>jar</packaging>

    <dependencies>
        <dependency>
            <groupId>${project.groupId}</groupId> ①
            <artifactId>hello-service-api</artifactId>
            <version>${project.version}</version> ①
        </dependency>
    </dependencies>
...
```

① Dependency references leveraging `${project}` variables module shares with dependency

Since we are using the same source tree, we can leverage `${project}` variables. This will not be the case when declaring dependencies on external modules.

2.5. Hello Service Interface

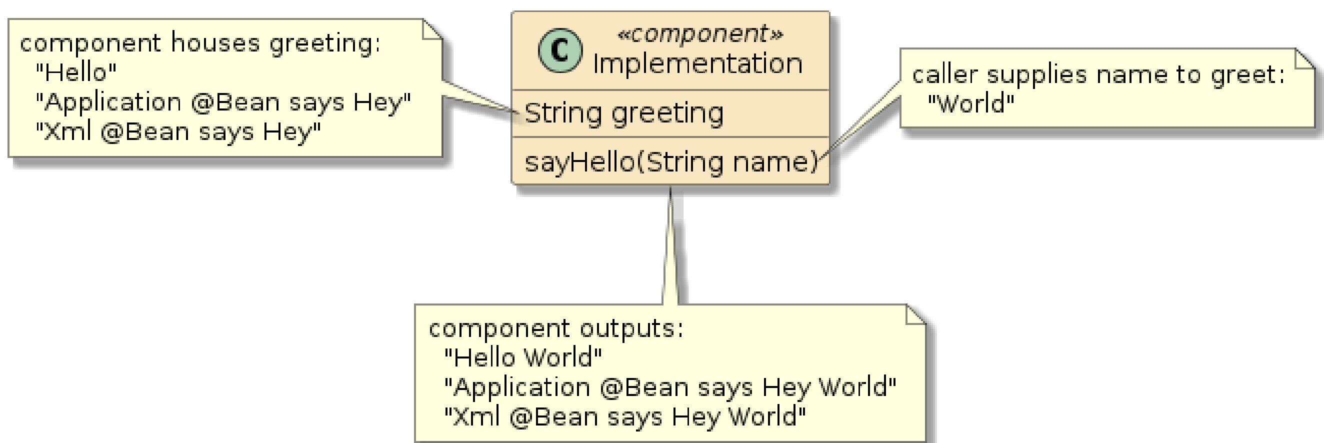
The interface is quite simple, just pass in the String name for what you want the service to say hello to.

```
package info.ejava.examples.app.hello;

public interface Hello {
    void sayHello(String name);
}
```

The service instance will be responsible for

- the greeting
- the implementation — how we say hello



2.6. Hello Service Sample Implementation

Our sample implementation is just as simple. It maintains the greeting in a final instance attribute and uses `stdout` to print the message.

```
package info.ejava.examples.app.hello.stdout; ①

public class StdOutHello implements Hello {
    private final String greeting; ②

    public StdOutHello(String greeting) { ③
        this.greeting = greeting;
    }
```

```
    @Override ④
    public void sayHello(String name) {
        System.out.println(greeting + " " + name);
    }
}
```

① Implementation defined within own package

② `greeting` will hold our phrase for saying hello and is made final to highlight it is required and will not change during the lifetime of the class instance

③ A single constructor is provided to define a means to initialize the instance. Remember — the `greeting` is final and must be set during class instantiation and not later during a setter.

④ The `sayHello()` method provides implementation of method defined in interface

`final` requires the value set when the instance is created and never change

Spring recommends constructor injection. This provides an immutable object and better assures that required dependencies are not null. [1]

2.7. Hello Service Modules Complete

We are now done implementing our sample service interface and implementation. We just need to build and install it into the repository to make available to the application.

2.8. Hello Service API Maven Build

```
$ mvn clean install -f hello-service-api
[INFO] Scanning for projects...
[INFO]
[INFO] -------------< info.ejava.examples.app:hello-service-api >--------------
[INFO] Building App::Config::Hello Service API 6.1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ hello-service-api ---
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:resources (default-resources) @ hello-service-
api ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory .../app-config/hello-service-
api/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ hello-service-api
---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to .../app-config/hello-service-api/target/classes
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:testResources (default-testResources) @ hello-
```

```
service-api ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory .../app-config/hello-service-
api/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ hello-
service-api ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ hello-service-api ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:3.1.2:jar (default-jar) @ hello-service-api ---
[INFO] Building jar: .../app-config/hello-service-api/target/hello-service-api-6.1.0-
SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:3.0.0-M1:install (default-install) @ hello-service-api
---
[INFO] Installing .../app-config/hello-service-api/target/hello-service-api-6.1.0-
SNAPSHOT.jar to .../.m2/repository/info/ejava/examples/app/hello-service-api/6.1.0-
SNAPSHOT/hello-service-api-6.1.0-SNAPSHOT.jar
[INFO] Installing .../app-config/hello-service-api/pom.xml to
.../.m2/repository/info/ejava/examples/app/hello-service-api/6.1.0-SNAPSHOT/hello-
service-api-6.1.0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.070 s
```

2.9. Hello Service StdOut Maven Build

```
$ mvn clean install -f hello-service-stdout
[INFO] Scanning for projects...
[INFO]
[INFO] ------------< info.ejava.examples.app:hello-service-stdout >------------
[INFO] Building App::Config::Hello Service StdOut 6.1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ hello-service-stdout ---
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:resources (default-resources) @ hello-service-
stdout ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory .../app-config/hello-service-
stdout/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ hello-service-
stdout ---
[INFO] Changes detected - recompiling the module!
```

```
[INFO] Compiling 1 source file to .../app-config/hello-service-stdout/target/classes
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:testResources (default-testResources) @ hello-
service-stdout ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory .../app-config/hello-service-
stdout/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ hello-
service-stdout ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ hello-service-stdout ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:3.1.2:jar (default-jar) @ hello-service-stdout ---
[INFO] Building jar: .../app-config/hello-service-stdout/target/hello-service-stdout-
6.1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:3.0.0-M1:install (default-install) @ hello-service-
stdout ---
[INFO] Installing .../app-config/hello-service-stdout/target/hello-service-stdout-
6.1.0-SNAPSHOT.jar to .../.m2/repository/info/ejava/examples/app/hello-service-
stdout/6.1.0-SNAPSHOT/hello-service-stdout-6.1.0-SNAPSHOT.jar
[INFO] Installing .../app-config/hello-service-stdout/pom.xml to
.../.m2/repository/info/ejava/examples/app/hello-service-stdout/6.1.0-SNAPSHOT/hello-
service-stdout-6.1.0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.658 s
```

[1] https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html_Constructor-based or setter-based DI?_,Spring.io

Chapter 3. Application Module

We now move on to developing our application within its own module containing two (2) classes similar to earlier examples.

```
|-- pom.xml
`-- src
    ``-- main
        `-- java
            `-- info
                `-- ejava
                    `-- examples
                        `-- app
                            `-- config
                                `-- beanfactory
                                    |-- AppCommand.java ②
                                    `-- SelfConfiguredApp.java ①
```

① Class with Java main() that starts Spring

② Class containing our first component that will be the focus of our injection

3.1. Application Maven Dependency

We make the Hello Service visible to our application by adding a dependency on the `hello-service-api` and `hello-service-stdout` artifacts. Since the implementation already declares a compile dependency on the interface, we can get away with only declaring a direct dependency just on the implementation.

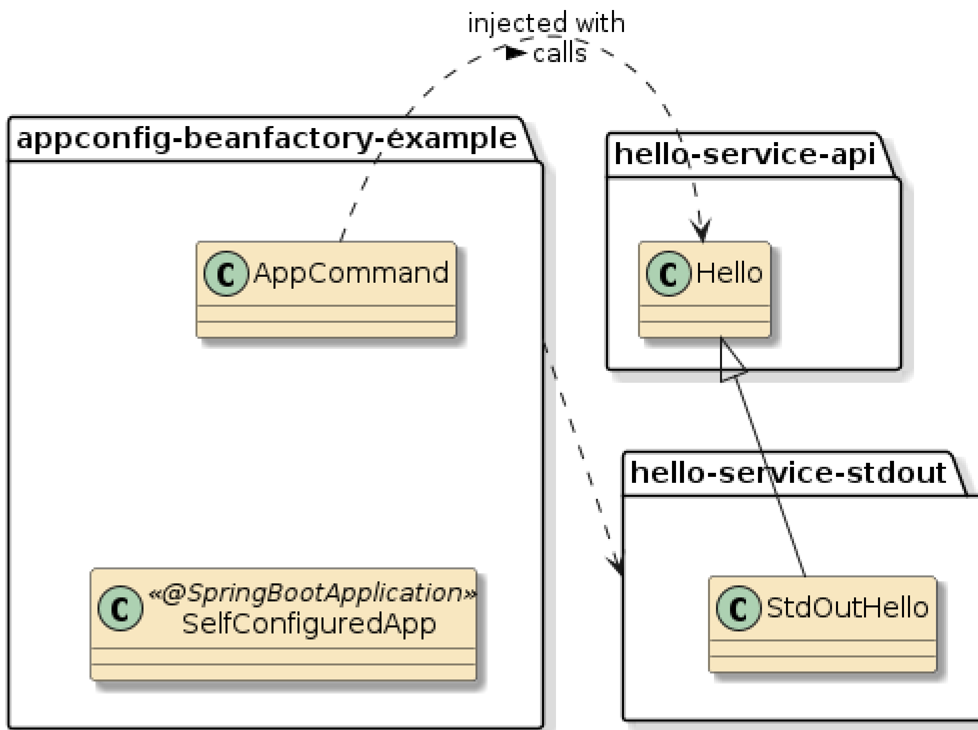
```
<groupId>info.ejava.examples.app</groupId>
<artifactId>appconfig-beanfactory-example</artifactId>
<name>App::Config::Bean Factory Example</name>

<dependencies>
  <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>hello-service-stdout</artifactId> ①
      <version>${project.version}</version>
  </dependency>
</dependencies>
```

① Dependency on implementation creates dependency on both implementation and interface

In this case, the module we are depending upon is in the same `groupId` and shares

the same version. For simplicity of reference and versioning, I used the ${project} variables to reference it. That will not always be the case.



3.2. Viewing Dependencies

You can verify the dependencies exist using the `tree` goal of the `dependency` plugin.

Artifact Dependency Tree

```
$ mvn dependency:tree -f hello-service-stdout
...
[INFO] --- maven-dependency-plugin:3.1.1:tree (default-cli) @ hello-service-stdout ---
[INFO] info.ejava.examples.app:hello-service-stdout:jar:6.1.0-SNAPSHOT
[INFO] \- info.ejava.examples.app:hello-service-api:jar:6.1.0-SNAPSHOT:compile
```

3.3. Application Java Dependency

Next we add a reference to the Hello interface and define how we can get it injected. In this case we are using contructor injection where the instance is supplied to the class through a parameter to the constructor.

The component class now has a non-default constructor to allow the `Hello` implementation to be injected and the Java attribute is defined as `final` to help assure that the value is assigned during the constructor.

```
package info.ejava.examples.app.config.beanfactory;

import org.springframework.boot.CommandLineRunner;
```

```java
import org.springframework.stereotype.Component;

import info.ejava.examples.app.hello.Hello;

@Component
public class AppCommand implements CommandLineRunner {
    private final Hello greeter; ①

    public AppCommand(Hello greeter) { ②
        this.greeter = greeter;
    }

    public void run(String... args) throws Exception {
        greeter.sayHello("World");
    }
}
```

① Add a reference to the Hello interface. Java attribute defined as `final` to help assure that the value is assigned during the constructor.

② Using contructor injection where the instance is supplied to the class through a parameter to the constructor

Chapter 4. Dependency Injection

Our `AppCommand` class has been defined only with the interface to `Hello` and not a specific implementation.

This Separation of Concerns helps improve modularity, testability, reuse, and many other desirable features of an application. The interaction between the two classes is defined by an interface.

But how do does our client class (`AppCommand`) get an instance of the implementation (`StdOutHello`)?

- If the client class directly instantiates the implementation — it is coupled to that specific implementation.

```
public AppCommand() {
    this.greeter = new StdOutHello("World");
}
```

- If the client class procedurally delegates to a factory — it runs the risk of violating Separation of Concerns by adding complex initialization code to its primary business purpose

```
public AppCommand() {
    this.greeter = BeanFactory.makeGreeter();
}
```

Traditional procedural code normally makes calls to libraries in order to perform a specific purpose. If we instead remove the instantiation logic and decisions from the client and place that elsewhere, we can keep the client more focused on its intended purpose. With this inversion of control (IoC), the application code is part of a framework that calls the application code when it is time to do something versus the other way around. In this case the framework is for application assembly.

Most frameworks, including Spring, implement dependency injection through a form of IoC.

Chapter 5. Spring Dependency Injection

We defined the dependency using the `Hello` interface and have three primary ways to have dependencies injected into an instance.

```
import org.springframework.beans.factory.annotation.Autowired;

public class AppCommand implements CommandLineRunner {
    //@Autowired -- FIELD injection ③
    private Hello greeter;

    @Autowired //-- Constructor injection ①
    public AppCommand(Hello greeter) {
        this.greeter = greeter;
    }

    //@Autowired -- PROPERTY injection ②
    public void setGreeter(Hello hello) {
        this.greeter = hello;
    }
```

① constructor injection - injected values required prior to instance being created

② field injection - value injected directly into attribute

③ setter or property injection - setter() called with value

5.1. @Autowired Annotation

The `@Autowired(required=…)` annotation

- may be applied to fields, methods, constructors
- `@Autowired(required=true)` - default value for `required` attribute
    - successful injection mandatory when applied to a property
    - specific constructor use required when applied to a constructor
        - only a single constructor per class may have this annotation
- `@Autowired(required=false)`
    - injected bean not required to exist when applied to a property
    - specific constructor an option for container to use
    - multiple constructors may have this annotation applied
        - container will determine best based on number of matches
    - **single constructor has an implied `@Autowired(required=false)`** - making annotation optional

There are more details to learn about injection and the lifecycle of a bean. However, know that we

are using constructor injection at this point in time since the dependency is required for the instance to be valid.

5.2. Dependency Injection Flow

In our example:

- Spring will detect the AppCommand component and look for ways to instantiate it
- The only constructor requires a Hello instance
- Spring will then look for a way to instantiate an instance of Hello

Chapter 6. Bean Missing

When we go to run the application, we get the following error

```
$ mvn clean package
...
***************************
APPLICATION FAILED TO START
***************************

Description:

Parameter 0 of constructor in AppCommand required a bean of type 'Hello' that could
not be found.

Action:

Consider defining a bean of type 'Hello' in your configuration.
```

The problem is that the container has no knowledge of any beans that can satisfy the only available constructor. The `StdOutHello` class is not defined in a way that allows Spring to use it.

6.1. Bean Missing Error Solution(s)

We can solve this in at least two (2) ways.

1. Add @Component to the StdOutHello class. This will trigger Spring to directly instantiate the class.

   ```
   @Component
   public class StdOutHello implements Hello {
   ```

   - problem: It may be one of many implementations of Hello

2. Define what is needed using a `@Bean` factory method of a `@Configuration` class. This will trigger Spring to call a method that is in charge of instantiating an object of the type identified in the method return signature.

   ```
   @Configuration
   public class AConfigurationClass {
       @Bean
       public Hello hello() {
           return new StdOutHello("...");
       }
   }
   ```

Chapter 7. @Configuration classes

`@Configuration` classes are classes that Spring expects to have one or more `@Bean` factory methods. If you remember back, our Spring Boot application class was annotated with `@SpringBootApplication`

```
@SpringBootApplication ①
//==> wraps @SpringBootConfiguration ②
//  ==> wraps @Configuration
public class SelfConfiguredApp {
    public static void main(String...args) {
        SpringApplication.run(SelfConfiguredApp.class, args);
    }
    //...
}
```

① `@SpringBootApplication` is a wrapper around a few annotations including `@SpringBootConfiguration`

② `@SpringBootConfiguration` is an alternative annotation to using `@Configuration` with the caveat that there be only one `@SpringBootConfiguration` per application

Therefore, we have the option to use our Spring Boot application class to host the configuration and the `@Bean` factory.

Chapter 8. @Bean Factory Method

There is more to `@Bean` factory methods than we will cover here, but at its simplest and most functional level — this is a series of factory methods the container will call to instantiate components for the application. By default they are all eagerly instantiated and the dependencies between them are resolved (if resolvable) by the container.

Adding a `@Bean` factory method to our Spring Boot application class will result in the following in our Java class.

```
@SpringBootApplication ④ ⑤
public class SelfConfiguredApp {
    public static void main(String...args) {
        SpringApplication.run(SelfConfiguredApp.class, args);
    }

    @Bean ①
    public Hello hello() { ②
        return new StdOutHello("Application @Bean says Hey"); ③
    }
}
```

① method annotated with `@Bean` implementation

② method returns `Hello` type required by container

③ method returns a fully instantiated instance.

④ method hosted within class with `@Configuration` annotation

⑤ `@SpringBootConfiguration` annotation included the capability defined for `@Configuration`

Anything missing to create instance gets declared as an input to the method and, it will get created in the same manner and passed as a parameter.

Chapter 9. @Bean Factory Used

With the `@Bean` factory method in place, all comes together at runtime to produce the following:

```
$ java -jar target/appconfig-beanfactory-example-*-SNAPSHOT-bootexec.jar
...
Application @Bean says Hey World
```

- the container
  - obtained an instance of a `Hello` bean
  - passed that bean to the `AppCommand` class' constructor to instantiate that `@Component`
- the `@Bean` factory method
  - chose the implementation of the `Hello` service (`StdOutHello`)
  - chose the greeting to be used ("Application @Bean says Hey")

    ```
    return new StdOutHello("Application @Bean says Hey");
    ```
- the AppCommand CommandLineRunner determined who to say hello to ("World")

  ```
  greeter.sayHello("World");
  ```

Chapter 10. Factory Alternative: XML Configuration

Although most developments today prefer Java-based configurations, the legacy approach of defining beans using XML is still available.

To do so, we define an `@ImportResource` annotation on a `@Configuration` class that references pathnames using either a class or file path. In this example we are referencing a file called `applicationContext.xml` in the `resources` package within the classpath.

```
import org.springframework.context.annotation.ImportResource;

@SpringBootApplication
@ImportResource({"classpath:contexts/applicationContext.xml"}) ①
public class XmlConfiguredApp {
    public static void main(String...args) {
        SpringApplication.run(XmlConfiguredApp.class, args);
    }
}
```

① `@ImportResource` will enact the contents of `context/applicationContext.xml`

The XML file can be placed inside the JAR of the application module by adding it to the `src/main/resources` directory of this or other modules in out classpath.

```
|-- pom.xml
`-- src
    `-- main
        |-- java
        |   `-- info
        |       `-- ejava
        |           `-- examples
        |               `-- app
        |                   `-- config
        |                       `-- xmlconfig
        |                           |-- AppCommand.java
        |                           `-- XmlConfiguredApp.java
        `-- resources
            `-- contexts
                `-- applicationContext.xml

$ jar tf target/appconfig-xmlconfig-example-*-SNAPSHOT-bootexec.jar | grep
applicationContext.xml
BOOT-INF/classes/contexts/applicationContext.xml
```

The XML file has a specific schema to follow. It can be every bit as powerful as Java-based configurations and have the added feature that it can be edited without recompilation of a Java

class.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean class="info.ejava.examples.app.hello.stdout.StdOutHello"> ①
        <constructor-arg value="Xml @Bean says Hey" />  ②
    </bean>
</beans>
```

① A specific implementation of ther `Hello` interface is defined

② Text is injected into the constructor when container instantiates

This produces the same relative result as the Java-based configuration.

```
$ java -jar target/appconfig-xmlconfig-example-*-SNAPSHOT-bootexec.jar
...
Xml @Bean says Hey World
```

Chapter 11. @Configuration Alternatives

With the basics of `@Configuration` classes understood, I want to introduce two other common options for a `@Bean` factory:

- scope
- proxyBeanMethods

By default,

- Spring will instantiate a single component to represent the bean (singleton)
- Spring will create a (CGLIB) proxy for each `@Configuration` class to assure the result is processed by Spring and that each bean client for singleton-scoped beans get the same copy. This proxy can add needless complexity depending on how sibling methods are designed.

The following concepts and issues will be discussed:

- shared (singleton) or unique (prototype) component instances
- (unnecessary) role of the Spring proxy (`proxyBeanMethods`)
- potential consequences of calling sibling @Bean methods over injection

11.1. Example Lifecycle POJO

To demonstrate, I created an example POJO that identifies its instance and tracks its component lifecycle.

- the Java constructor is called for each POJO instance created
- `@PostConstruct` methods are called by Spring after dependencies have been injected. If and when you see debug from the `init()`, you know the POJO has been made into a component, with the potential for Spring container interpose.

Example POJO Class Created and Used by @Bean Factory

```
public class Example {
    private final int exampleValue;
    public Example(int value) { this.exampleValue = value; } ①
    @PostConstruct ②
    void init() {
        System.out.println("@PostConstruct called for: " + exampleValue);
    }
    public String toString() { return Integer.toString(exampleValue); }
}
```

① Constructor will be called for every instance created

② `@PostConstruct` will only get called by when POJO becomes a component

11.2. Example Lifecycle @Configuration Class

To further demonstrate, I have added a `@Configuration` class with some options that will be changed during the example.

Example @Configuration and Source @Bean Factory

```
@Configuration //default proxyBeanMethods=true
//@Configuration(proxyBeanMethods = true)
//@Configuration(proxyBeanMethods = false)
public class BeanFactoryProxyBeansConfiguration {
    private int value = 0;

    @Bean //default is singleton
    //@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON) //"singleton"
    Example bean() {
        return new Example(value++);
    }
```

11.3. Consuming @Bean Factories within Same Class

Thre are two pairs of `Example`-consuming `@Bean` factories: calling and injected.

- Calling `@Bean` factories make a direct call to the supporting `@Bean` factory method.
- Injected `@Bean` factories simply declare their requirement in method input parameters.

Example @Bean Consumers

```
    @Bean ①
    String calling1()             { return "calling1=" + bean(); }
    @Bean ①
    String calling2()             { return "calling2=" + bean(); }
    @Bean ②
    String injected1(Example bean) { return "injected1=" + bean; }
    @Bean ②
    String injected2(Example bean) { return "injected2=" + bean; }
```

① calling consumers call the sibling `@Bean` factory method directly

② injected consumers are passed an instance of requirements when called

11.4. Using Defaults (Singleton, Proxy)

By default:

- `@Bean` factories use singleton scope
- `@Configuration` classes use `proxyBeanMethods=true`

That means that:

- @Bean factory method will be called only once by Spring
- `@Configuration` class instance will be proxied and direct calls (`calling1` and `calling2`) will receive the same singleton result

Proxied @Configuration Class Emits Same Singleton to all Consumers

```
@PostConstruct called for: 0 ②
calling1=0 ①
calling2=0 ①
injected1=0 ①
injected2=0 ①
```

① only one POJO instance was created

② only one component was initialized

11.5. Prototype Scope, Proxy True

If we change component scope created by the `bean()` method to "prototype"...

@Bean factory Scope set to non-default "prototype"

```
@Bean
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE) //"prototype"
Example bean() {
    return new Example(value++);
}
```

...we get a unique POJO instance/component for each consumer (calling and injected) while `proxyBeanMethods` is still `true`.

Different Instance Produced for each Consumer and all Consumer Types

```
@PostConstruct called for: 0 ②
@PostConstruct called for: 1 ②
@PostConstruct called for: 2 ②
@PostConstruct called for: 3 ②
calling1=0 ①
calling2=1 ①
injected1=2 ①
injected2=3 ①
```

① unique POJO instances were created

② each instance was a component

11.6. Prototype Scope, Proxy False

If we drop the CGLIB proxy, our configuration instance gets lighter, but ...

Spring Proxy Off, Prototype Scope

```
@Configuration(proxyBeanMethods = false)
public class BeanFactoryProxyBeansConfiguration {
    @Bean @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE) //"prototype"
    Example bean() {
```

...only injected consumers are given "component" beans. The "calling" consumers are given "POJO" beans, that lack the potential for interpose.

```
@PostConstruct called for: 2 ②
@PostConstruct called for: 3 ②
calling1=0 ①
calling2=1 ①
injected1=2 ①
injected2=3 ①
```

① each consumer is given a unique instance

② only the injected callers are given components (with interpose potential)

11.7. Singleton Scope, Proxy False

Keeping the proxy eliminated and reverting back to the default singleton scope for the bean ...

Spring Proxy Off, Singleton Scope

```
@Configuration(proxyBeanMethods = false)
public class BeanFactoryProxyBeansConfiguration {

    @Bean @Scope(ConfigurableBeanFactory.SCOPE_SINGLETON) //"singleton" - default
    Example bean() {
```

...shows that only the injected consumers are receiving a singleton instance — initialized as a component, with the potential for interpose.

```
@PostConstruct called for: 0 ③
calling1=1 ②
calling2=2 ②
injected1=0 ①
injected2=0 ①
```

① injected consumers get the same instance

② calling consumers get unique instances independent of `@Scope`

③ only the injected consumers are getting a component (with interpose potential)

11.8. @Configuration Takeaways

- Spring instantiates all components, by default, as singletons — with the option to instantiate unique instances on demand when `@Scope` is set to "prototype".
- Spring, by default, constructs a CGLIB proxy to enforce those semantics for both calling and injected consumers.
  - Since `@Configuration` classes are only called once at start-up, it can be a waste of resources to construct a CGLIB proxy.
    - Using injection-only consumers, with no direct calls to `@Configuration` class methods, eliminates the need for the proxy.
    - adding `proxyFactoryBeans=false` eliminates the CGLIB proxy. Spring will enforce semantics for injected consumers

Chapter 12. Summary

In this module we

- decoupled part of our application into three Maven modules (app, iface, and impl1)
- decoupled the implementation details (`StdOutHello`) of a service from the caller (`AppCommand`) of that service
- injected the implementation of the service into a component using constructor injection
- defined a `@Bean` factory method to make the determination of what to inject
- showed an alternative using XML-based configuration and `@ImportResource`
- explored the differences between calling and injected sibling component consumers

In future modules we will look at more detailed aspects of Bean lifecycle and @Bean factory methods. Right now we are focused on following a path to explore decoupling our the application even further.