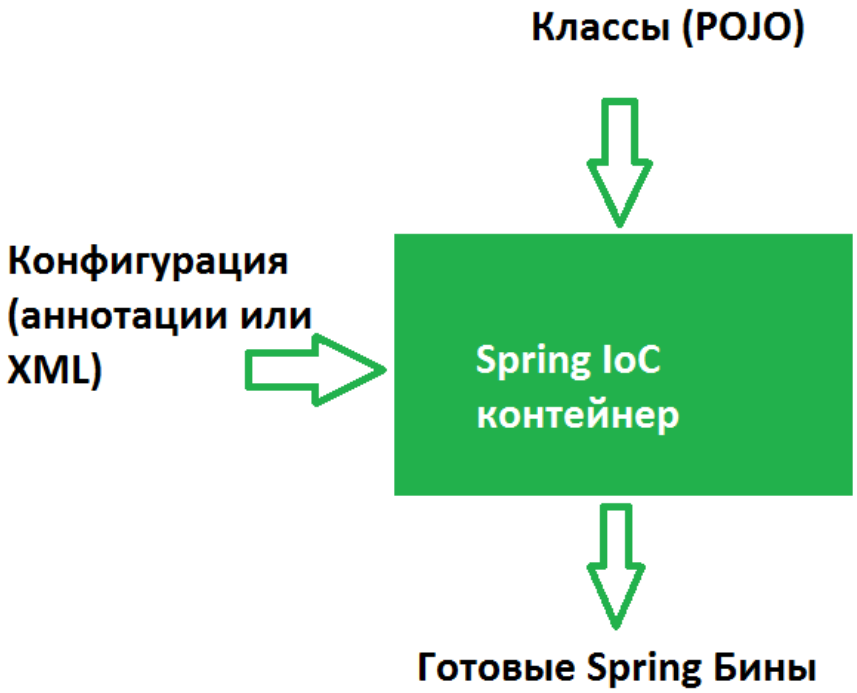
Spring Framework: инверсия управления и внедрение зависимостей

1. Инверсия управления и внедрение зависимостей.

До того как перейти к написанию программы, разберём, что такое иверсия управления (Inversion of control):

Ключевая особенность приложения, написанного на Spring, состоит в том что большую часть объектов создаем не мы, а Spring. Мы лишь конфигурируем классы (с помощью аннотаций либо в конфигурационном XML), чтобы «объяснить» фреймворку Spring, какие именно объекты он должен создать за нас, и полями каких объектов их сделать. Spring управляет созданием объектов и потому его контейнер называется IoC-контейнер. IoC расшифровывается как Inversion of Control. А объекты, которые создаются контейнером и находятся под его управлением, называются бинами.

Иллюстрировать это можно так:



Внедрение зависимости (Dependency injection) — ключевой шаблон проектирования в Spring. Мы говорим фреймворку создать за нас бины (иначе говоря — объекты) и внедрить их в другие бины. И фреймворк это делает.

Но как объяснить фреймворку Spring, что такой-то бин должен стать зависимостью для другого бина? Вариантов немного, а самых частых всего

два: бин внедряется либо через конструктор класса, либо с помощью сеттера. Первое называется constructor-based injection, а второе — setter-based injection.

Чтобы внедрить бин, классов нам недостаточно, Spring имеет дело с бинами, а не классами. Поэтому нужно сконфигурировать эти классы так, чтобы Spring контейнер создал на их основе бины. В конфигурации заодно будут заданы и рзависимости. Конфигурировать бины можно либо с помощью аннотаций, либо с помощью XML. (Но учтите, что XML-конфигурация немного устарела.)

2. Создание проекта.

Для того, что бы собрать проект под spring мы можем обратится к сайту Spring Initializer:
https://start.spring.io/
и сгенерировать проект. В данном примере нам не нужны сторонние зависимости. Project выбираем Maven. Язык java. Версия Spring Boot 2.6.6. Данные проекта можно не измениять. Пакет jar и версия java 17.

2. Конфигурация бинов с помощью XML.
Обратимся к файлу applicationContext.xml, этот файл должен быть создан нами самостоятельно и помещён в папку «resources». Его содержимое:

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

    <context:annotation-config/>
    <context:component-scan base-package="com.baeldung.applicationcontext"/>




    <bean id="table"
        class="com.springboot.spboot.objectsDatabase.Table">
        <property name="name" value="Clients"/>
    </bean>
    <bean id="database"
        class="com.springboot.spboot.Database">
        <constructor-arg ref="table"/>
        <property name="name" value="Store"/>
    </bean>
</beans>
```

Из этого файла spring создаст два объекта «table» и «database». В начале нам необходимо прописать xml конфигурацию, её запоминать не нужно, она везде одинакова.

Создаём бин, указываем, что его id (уникальный идентификатор объекта), присваиваю ему знаачение «table» и прописываю путь к классу Table.

```
    <bean id="table"
        class="com.springboot.spboot.objectsDatabase.Table">
```

Инициализируем поле класса tabel через setter в setName. Для этого мы используем тэг property в name передаём имя сеттера отбразывая set и прописывая Name с нижним регистром. В value указываем значение передаваемое сеттеру.

```
<property name="name" value="Clients"/>
```

Следующему бину, при создании объекта в конструктор передаю бин «table», это значит, что при создании объекта database ему будет передан в качестве зависимости объект table (по ссылке), это и есть внедрение зависимостей.

```
  <bean id="database"
        class="com.springboot.spboot.Database">
        <constructor-arg ref="table"/>
```

Инициализируем поле класса database:

```
<property name="name" value="Store"/>
```

3. Классы проекта:
Класс «Table»:

```
package com.springboot.spboot.objectsDatabase;

public class Table {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Table{" +
                "nameTable='" + name + '\'' +
                ", hashCode='" + hashCode() +'\'' +
                '}';
    }
}
```

Класс «Database»:

```
package com.springboot.spboot;

import com.springboot.spboot.objectsDatabase.Table;

public class Database {
    private Table table;
    private String name;
    Database(){}
    Database(Table table) {
        //конструктор принимает объект table
        this.table = table;
    }
    public void setTable(Table table) {
```

```
        //setter для объекта table
        this.table = table;
    }
    public void setName(String name) {
        // текстовое поле
        this.name = name;
    }
    @Override
    public String toString() {
        return "Database{" +
                "table=" + table +
                ", name='" + name + '\'' +
                ", hashCode='" + hashCode() +'\'' +
                '}';
    }
}
```

4. Класс с методом main

```
package com.springboot.spboot;

import com.springboot.spboot.objectsDatabase.Table;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.support.ClassPathXmlApplicationContext;

@SpringBootApplication
public class SpbootApplication {

	public static void main(String[] args) {
		ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
				"applicationContext.xml"
		);
		Database databaseFirst = context.getBean("database", Database.class);
		System.out.println(databaseFirst);
		context.close();
	}

}
```

Обращаемся к конфигурационному файлу. Класс ClassPathXmlApplicationContext мы получаем spring context. Он обращается к файлу applicationContext, считывает его и помещает все бины, которые там описаны в application context. applicationContext должен лежать в папке «resources».

```
		ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
				"applicationContext.xml"
```

Достаём бин из контекста. В качестве аргумента мы должны указать id бина и класс, бин которого мы хотим получить.

```
Database databaseFirst = context.getBean("database", Database.class);
```

Выводим объект на экран:

```
System.out.println(databaseFirst);
```

Обязательно закрываем контекст:

```
context.close();
```

Результат:

```
Database{table=Table{nameTable='Clients', hashCode='73181251'}, name='Store', hashCode='1946645411'}
```

Спринг создал бин table и database, передал table в конструктор database, инициализировал поля и вернул нам объект database.

5. Передача бина table в database через setter

Заменяем следующий код:

```
<constructor-arg ref="table"/>
```

на:

```
<property name="table" ref="table"/>
```

Заменяем value на ref и прописываем id бина.

Результат:

```
Database{table=Table{nameTable='Clients', hashCode='396883763'}, name='Store', hashCode='1781241150'}
```

6. Внедрение значений из внешнего файла

Создадим для этого отдельный файл, который назовём «application.properties», создать его необходимо в папке «resources».

Содежание файла:

```
database.name=Store
```

Присваеваем table.name значение Products a database.name значение Store.

Далее заменяем:

```
<property name="name" value="Store"/>
```

на:

```
<property name="name" value="${database.name}"/>
```

И в applicationContext.xml до бинов вставляем следующий код:

```
<context:property-placeholder location="classpath:application.properties"/>
```

Мы указываем где хранится наш файл.

Запуск, результат:

```
Database{table=Table{nameTable='Clients', hashCode='396883763'}, name='Store', hashCode='1781241150'}
```

Значение через файл присвоилось объекту класса database.

7. Область видимости бинов scope

Scope задаёт то, как спринг будет создавать бины.

Есть такой scope, который называется Singleton, он используется по умолчанию. Это означает, что spring создаст только один объект и далее при вызове getBean() вернётся ссылка на этот объект. И как в роли зависимость для других бинов будет присваиваться ссылка на один объект.

Второй scope это Prototype. Он каждый раз создаёт новый объект при вызове getBean() и возвращает его ссылку.

Внесём некоторые изменения в код:

Вставим следующую строку в бин datatype:

```
scope="prototype">
```

Должно выйти так:

```
<bean id="database"
      class="com.springboot.spboot.Database"
      scope="prototype">
    <property name="table" ref="table"/>
    <property name="name" value="${database.name}"/>
</bean>
```

Далее в метод main добавим следующие строки:

```
Database databaseSecond = context.getBean("database", Database.class);
System.out.println(databaseSecond);
```

после объекта databaseFirst.

Запуск, результат:

```
Database{table=Table{nameTable='Clients', hashCode='1995250556'}, name='Store', hashCode='793138072'}
Database{table=Table{nameTable='Clients', hashCode='1995250556'}, name='Store', hashCode='1062635358'}
```

Обратим внимания на hashCode объектов. У двух объектов table hashCode одинаков, следовательно, мы можем утверждать (с большей долей вероятности), что это один и тот же объект. Это из за того, что у нас установлен scope = singleton.

Однако, объекты database имеет разные hashCode, они точно разные, это дело рук scope = prototype.