**Spring**

**Spring Framework** – в общем приближении это **контейнер IoC** и **набор модулей**, отвечающих за некоторый функционал, используя которые мы можем построить приложение любой сложности.

Причем совсем не обязательно подтягивать все модули – основная идея в том, что модули можно использовать независимо от всех остальных... они обладают независимым законченным функционалом.

Для того, чтобы прочувствовать **Spring** – будем проект создавать постепенно. Не будем пользоваться оснастками идеи и инициализатором с сайта спринга. Потому, что они предлагают уже готовые сборки и не позволяют подключать модули по отдельности... Мы будем постепенно рассматривать модули.

Создадим просто maven проект и рассмотрим первый модуль core. Это основной модуль и он на сайте <https://spring.io/projects> называется **Spring Framework** – базовый модуль, который используется всеми остальными модулями. По сути он и представляет **контейнер IoC.**

Для того, чтобы подгрузить этот модуль нужно добавить к нам в pom-ку

**<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-core</artifactId>  
 <version>5.3.29</version>  
</dependency>**

Спринг позволяет разработчикам избавится от рутинных действий и сосредоточится на действительно важных задачах по реализации функционала задачи. Относительно нашей последней задачи – не нужно думать, что нужно где-то создать объект **dao**, создать объекты сервисов в контроллерах и т.д. Да и даже в реализации **dao** мы постоянно делали одни и теже вещи в методах... **Спринг это делает за нас.**

У спринга есть свой контейнер в котором он создает объеты, которые нам будут нужны для работы, и когда нам понадобятся эти объекты, то мы берем из этого контейнера (дао, сервисы). **При этом мы их даже не создаем.**

Нам необходимо просто указать какие объеты нам нужны, дать их описание (проперти, локальные поля (например поле дао в сервисе) ) и их связи.

**Core** часть и занимается описанием этих контейнеров и объектов. **В спринге объекты в контейнере называются бинами (bean)**. И когда мы достаем из контейнера эти объекты – они превращаются в java объекты.

**Чтобы начать полноценную работу, одного модуля недостаточно,** необходимо подгрузить еще один модуль – **Spring Context** – это нужно для работы спрингового контейнера.

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-context</artifactId>**

**<version>5.3.29</version>**

**</dependency>**

Что самое интересное, spring core есть в зависимостях у **Spring Context**... **поэтому core можем удалить из помки**... он подтянется как зависимость контекста.

**Итак для начала работы необходимо описать контейнер**... для этого существуют формы:

* xml – просто файл в котором этот спринговый объект описан в виде xml
* java – создается класс, в котором описывается этот контейнер, его содержимое
* аннотационный подход...

**1**  В каталоге **resources** создадим файл new -> XML Configuration File -> Spring Config -> **01\_helloworld.xml**

Это файл, который соответствует определенным схемам, которые определяют набор тегов, использующихся в файле. Плюсы этого подхода – все описание в одном месте и на виду и главное – **можем описать сложную структуру взаимосвязей**... Минусы – xml это тупо простыня букв.

Опишем контейнер для первой программы... как всегда это будет программа **Hello world**.

Итак.... бины в xml описываются с помощью тэга <bean>, у него есть ряд свойств:

* **id** – уникальный идентификатор
* **name** – тоже идентификатор, но не уникальный
* **class** – показывает в какой класс должен превратиться бин...

**<?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">  
  
 1 <bean name="default-message" class="java.lang.String"/>**

**2 *<bean name="message" class="java.lang.String">***

***<constructor-arg value="Hello world"/>***

***</bean>*</beans>**

Здесь мы определили 2 бина с именем **default message** и **message** и указали контейнеру, что эти бины должны превратиться в класс **String**.

Напишем класс **Main** с методом **main** – определим точку входа.

В нашем методе нам сначала необходимо получить реализацию нашего контейнера. Сушествует несколько способов получения контейнера в зависимости от метода описания... конкретно для xml – через класс **ClassPathXmlApplicationContext**.

И уже потом из этого контекста можно получать бины.... как вручную, так и автоматически.

Если мы не дадим никакого значения, то **Spring** найдет у класса дефолтовый конструктор и создат объект ( у строки это пустая строка...), но мы можем задать параметры с помощью специального тега **constructor-arg** причем этот тег имеет несколько аттрибутов (**index, name, value…**) и в зависимости от этих значений будет выбираться соответствующий конструктор этого класса.

**public class Main {**

**public static void main(String[] args) {**

**AppContext ctx = new ClassPathXmlApplicationContext("01\_helloworld .xml");**

**1 String message = (String) ctx.getBean("default-message");**

**2 String message = (String) ctx.getBean("message");**

**System.out.println(message);**

**}**

**}**

Это простейшая програма на **Spring**, немного усложним задачу.... создадим новый пакет **\_02\_car**. На уровне этого пакета создадим пакет **entities.car**, в котором создадим класс **Car.**

В классе определим **2 поля**, определим конструкторы: **по умолчанию** (в нем определим дефолтное значение модели), **с одним параметром** ( model ), и **2 конструктора с 2-мя параметрами** (поменяем параметры местами), определим геттеры и сеттеры и toString

**public class Car {  
 private int id;  
 private String model;  
  
 public Car() {  
 model = "Default model";  
 }**

**. . .**

**}**

**Опишем наш контейнер** (созддим файл конфигурации **02\_car.xml**) в котором определим:

*Bean, при построении которого будет использован конструктор с одним параметром*

**<bean name="car1" class="by.itclass.entities.car.Car">**

**<constructor-arg value="BelGee"/>**

**</bean>**

*Bean, при построении которого используем кон-р по умолчанию... но устанавливаем значение другого поля...*

**<bean name="car2" class="by.itclass.entities.car.Car">**

**<property name="id" value="1"/>**

**</bean>**

*Гораздо интереснее использование двух параметров... ведь у нас таких конструктора 2 штуки а значения мы пишем в xml как строки... для такого случая у этого параметра есть* ***аттрибуты*** *для однозначной идентификации:*

*В этом случае выберется конструктор в зависимости от порядкового номера поля*

**<bean name="car3" class="by.itclass.entities.car.Car">**

**<constructor-arg index="0" value="3"/>**

**<constructor-arg index="1" value="WV"/>**

**</bean>**

*А в этом случае в конструктор значения будут подставляться по именам полей*

**<bean name="car4" class="by.itclass.entities.car.Car">**

**<constructor-arg name="id" value="3"/>**

**<constructor-arg name="model" value="WV"/>**

**</bean>**

Создадим точку входа – в пакете \_02\_car создадим Main класс с main методом.

**public static void main(String[] args) {**

**AppContext ctx = new ClassPathXmlApplicationContext("02\_car.xml");**

**List<Car> cars = new ArrayList<>();**

**cars.add((Car) ctx.getBean("car1"));**

**cars.add((Car) ctx.getBean("car2"));**

**cars.add((Car) ctx.getBean("car3"));**

**cars.add((Car) ctx.getBean("car4"));**

**cars.forEach(System.out::println);  
}**

Но опять же.... это простые объекты, в которых только поля... что если нам нужно создать сложный объект, используя композицию?

Создадим рядом с нашим классом **Car** еще один, который назовем **Engine**… определим в нем поле строковое **type**, создадим конструктор с параметром.

**public class Engine {**

**private String type;**

**}**

Добавим его в наш класс **Car** как поле, добавим **конструктор с тремя параметрами и добавим вывод типа двигателя в toSring**

**public class Car {**

**private int id;**

**private String model;**

**private Engine engine;**

**public Car(int id, String model, Engine engine) {**

**this.id = id;**

**this.model = model;**

**this.engine = engine;**

**}**

**}**

В нашем **старом подходе эти две сущности были жестко связаны**.. сначала мы создавали мотор, потом создавали машину, используя созданный объект мотора...

Сейчас же... нас не волнует создание мотора... когда мы скажем контейнеру достать машину с определенным мотором он сам мотор создаст и засунет его в эту машину... **если мы это опишем в контейнере**

**<bean name="engine-gas" class="by.itclass.entities.car.Engine">**

**<constructor-arg value="Gasoline"/>**

**</bean>**

**<bean name="car5" class=" by.itclass.entities.car.Car">**

**<constructor-arg name="id" value="3"/>**

**<constructor-arg name="model" value="Ferrari"/>**

**<constructor-arg name="engine" ref="engine-gas"/>**

**</bean>**

**Car car = (Car) ctx.getBean("car5");**

**System.out.println(car);**

***МЫ НЕ ЗАМОРАЧИВАЕМСЯ СОЗДАНИЕМ ОБЪЕКТОВ... НИРАЗУ НЕ ВЫЗВАЛИ NEW()***

***Еще одно интересное свойство...*** создадим конфиг (файл – **03\_scope.xml**), в котором определиим один бин... а в Main методе получим несколько экземпляров. Мы не переопределяли метод хэшкод и **ожидаем, что будут созданы разные объекты с разным хэшем.**..

**<bean name="car" class=" by.itclass.entities.car.Car"/>**

Создадим новый пакет \_03\_scope, в котором создадим класс с новой токой входа..

**public static void main(String[] args) {**

**AppContext ctx = new ClassPathXmlApplicationContext("03\_scope.xml");  
 Car car1 = (Car) ctx.getBean("car");  
 Car car2 = (Car) ctx.getBean("car");  
 Car car3 = (Car) ctx.getBean("car");  
  
 System.out.println(car1.hashCode());  
 System.out.println(car2.hashCode());  
 System.out.println(car3.hashCode());  
}**

**Но.... нет!**

За уникальность объектов отвечает свойство **scope**, которое имеет несколько значений:

* **prototype**
* **singleton** (по умолчанию) в консольном только 2

И только когlа мы выставим скоуп = **prototype** – то создадутся разные объекты!!!

**<bean name="car" class="by.itclass.entities.car.Car" scope="prototype"/>**

***Еще один атрибут бина – autowire.*** Определим в нашем классе **Car** конструктор **только с мотором**. Создадим конфигурацию (файл **autowire.xml**)

**<bean name="engine-gas" class=" by.itclas.entities.car.Engine">  
 <constructor-arg value="Gasoline"/>  
</bean>  
  
<bean name="car" class="by.itclas.entities.car.Car" autowire="constructor"/>**

Autowire = **constructor**, **byType**, **byName** – гибкость.

**public static void main(String[] args) {**

**AppContext ctx = new ClassPathXmlApplicationContext("04\_autowire.xml");**

**Car car = (Car) ctx.getBean("car");**

**System.out.println(car);**

**}**

**2 Java-based способ, основан на создании специального класса конфигурации.**

Создадим пакет **\_05\_java\_config** и в нем создадим класс **AppConfig**

**public class AppConfig {  
 @Bean("message")  
 public String message() { return "Hello world"; }  
  
 @Bean  
 public Car car1() { return new Car(); }  
  
 @Bean  
 public Car car2() { return new Car("Peugeot"); }  
  
 @Bean  
 public Engine engine() { return new Engine("Gasoline"); }  
  
 @Bean  
 public Car car3() { return new Car(engine()); }  
}**

При этом подходе бины описываются в виде методов, которые возвращают объекты какого-либо класса, причем **имя метода соответствует имени бина**, который мы хотим получить.

**AppContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);  
String message = (String) ctx.getBean("message");  
System.out.println(message);  
Car car1 = (Car) ctx.getBean("car1");  
Car car2 = (Car) ctx.getBean("car2");  
Car car3 = (Car) ctx.getBean("car3");  
List<Car> cars = Arrays.asList(car1, car2, car3);  
cars.forEach(System.out::println);**

Немного усложним...попробуем добавить коллекцию. Создадим в **entities.libraries** класс **Library** и **Book**.

**public class Library {**

**private int id;**

**//@Autowired – над полем не рекомендуется. или конструктор или над сеттером**

**private List<Book> books;**

**public Libraries(int id) {**

**this.id = id;**

**}**

***Геттеры и сеттеры для всех... и toString()***

**@Autowired**

**public void setBooks(List<Book> books) {**

**this.books = books;**

**}**

**}**

**public class Book {**

**private String title;**

***конструктор и toString()***

**}**

Создадим новую конфигурацию

**public class AppConfigList {  
  
 @Bean  
 public List<Book> books() {  
 return Arrays.asList(new Book("Title1"), new Book("Title2")); }  
  
 @Bean  
 public Library library() { return new Library(1); }  
}**

А также новую точку входа..

**AppContext ctx = new AnnotationConfigApplicationContext(AppConfigList.class);**  
**Library library = ctx.getBean(Library.class); //так тоже можно  
System.out.println(library);**

Опять же... с моей точки зрения ни разу не удобный способ... просто мы вместо xml пишем методы... но это хоть немного похоже на программирование... мы тут в поте лица классы создаем да методы в этих классах пишем. А по факту – та же самая простыня букв.

**3 Annotation configuration** способ

Для объяснения того, что уже рассказалось, поступим следующим способом... в пакете **entities.mvcbeans** создадим классы **Controller**, **Service,** **Dao и ConnectionManager**… В этих классах просто переопредели **toString()**, таким образом, чтобы он нам выводил имя класса.

**public class ConnectionManager {  
 @Override  
 public String toString() { return "ConnectionManager class"; }  
}**

**public class Dao {  
 @Override  
 public String toString() { return "Dao class"; }  
}**

**public class Service {  
 @Override  
 public String toString() { return "Service class"; }  
}**

**public class Controller {  
 @Override  
 public String toString() { return "Controller class"; }  
}**

Над классами, повесим аннотацию **@Component**... и в этом случае во время анализа кода когда **Spring** увидит эту аннотацию - он поймет, что нужно создать бин и поместить его в контейнер.

Создадим в пакете **\_06\_annotation** новый класс **Main**, в нем определим контекст. А вот для того, чтобы это работало – контекст создадим не из некоего файла, а укажем наши классы бинов. (**также можно указать путь к каталогу…** )

**AppContext ctx = new AnnotationConfigApplicationContext(  
 ConnectionManager.class, Controller.class, Service.class, Dao.class);**

***AppContext ctx = new AnnotationConfigApplicationContext(***

***"by.itclass.entities.mvcbeans");***

**ConnectionManager cn = ctx.getBean(ConnectionManager.class);  
Controller controller = ctx.getBean(Controller.class);  
Service service = ctx.getBean(Service.class);  
Dao dao = ctx.getBean(Dao.class);  
  
System.out.println(cn);  
System.out.println(dao);  
System.out.println(service);  
System.out.println(controller);**

Мы получили наши объекты... также мы уже знаем, что наши бины лежат в контейнере и мы спокойно можем их связать между собой... В контроллере сервис инициализировался в init() методе, в сервисе dao было полем и инициализировалось в конструкторе... и т.д.

Для того, чтобы связать эти объекты, нужно просто добавить сеттеры в класс и добавить аннтотацию **@Autuwired**..

**private ConnectionManager connectionManager;  
  
@Autowired  
public void setConnectionManager(ConnectionManager connectionManager) {  
 this.connectionManager = connectionManager; }  
  
@Override  
public String toString() {  
 return "Dao class, Connection Manager : " + connectionManager;  
}**

-------------------------------------------------------------------------------------------------------------------------------

**private Dao dao;  
  
@Autowired  
public void setDao(Dao dao) {**

**this.dao = dao; }  
  
@Override  
public String toString() {  
 return "Service class, Dao : " + dao;  
}**

**private Service service;  
  
@Autowired  
public void setService(Service service) {**

**this.service = service; }**

**@Override  
public String toString() {  
 return "Controller class, Service : " + service;  
}**

А в main классе просто выведем контроллер.

**AppContext ctx = new AnnotationConfigApplicationContext(**

**"by.itclass.entities.mvcbeans");  
Controller controller = ctx.getBean(Controller.class);**

И теперь... когда уже немного понятней... или не понятней, перейдем к **основнвным понятиям Spring**

* **IoC (Inversion of Control)** – при простом Java подходе мы сначала должны были создать окружение - **объекты** которые могут взаимодействовать между собой, используя методы. **IoC утверждает, что это все окружение нам создаст контейнер Spring**.
* **DIP (Dependency Injection Principle) -** набор правил, которые позволяют реализовать идею IoC.
* **DI (Dependency Injection)** - это непосредственная реализация **DIP** (**autowire**… и не только).

Одно из правил DIP – **модули верхнего уровня не должны быть жестко связаны с модулями нижнего уровня**. Это правило у нас нарушалось... В **Service** (верхний уровень) был жестко определен через **new()** модуль **DAO** (нижний уровень).

Чтобы этого избежать – есть второе правило – **все должно быть постороено на абстракциях (абстракция не должна зависить от деталей, а детали должны зависеть от абстракции) т.е. использовать интерфейсы или абстрактные классы**. В сервисе использовать поле не класса, а интерфейса.. и это поле инициализировать не в конструкторе по умолчанию, а передавая его как аргумент или используя сеттер.