Spring Framework

Spis treści

1. What is what and why

There is a project called *Spring framework*. This project consists of several parts (for instance, when we add Spring Framework as a dependency to our project, it adds several jars). Some of these parts are:

- spring-core – it is an inversion of control container
- spring-webmvc – it is a framework for web applications
- spring-webflux – it is a new (since Spring Framework 5.0, so since 2017), reactive framework for web applications

Authors of *Spring Framework* have also created a lot of other projects which names start with *Spring*: for instance *Spring Boot* or *Spring Security*. It causes some ambiguity: when people talk about *Spring,* sometimes they mean *Spring Framework*, sometimes they mean all *Spring something* projects and sometimes they mean I don't know what.

This document describes two parts of Spring Framework: spring-core and spring-webmvc.

2. Why do we need Spring

The program I write in this chapter and in following chapters is in the project *MySpringHelloWorld*.

Let's imagine I have to write a program which will show current date.

I will make it modular. There will be:

- DateProducer – a factory that can produce a date,
- DatePrinter – a class that can print out a date,
- MyProgram – a class that needs to have injected DateProducer and DatePrinter and that uses them to print a date,
- Main – a class that instantiates all three classes, injects the instance of DateProducer and DatePrinter into MyProgram and runs MyProgram.

I create in NetBeans a new project of type *maven -> Java application*. I call it *MySpringHelloWorld*. In it, I create the *DateProducer* class:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
public class DateProducer {
    private Date date;
    public void setDate(Date date) {
```

```
        this.date = date;
    }
    public Date giveMeADate() {
        return date;
    }
}
```

And the *DatePrinter* class:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
public class DatePrinter {
    public void printDate(Date d) {
        System.out.println("date: " + d);
    }
}
```

And the *MyProgram* class:

```
package pl.kuku.myspringhelloworld;
public class MyProgram {
    private DateProducer producer;
    private DatePrinter printer;
    public void setProducer(DateProducer producer) {
        this.producer = producer;
    }
    public void setPrinter(DatePrinter printer) {
        this.printer = printer;
    }
    public void run() {
        printer.printDate(producer.giveMeADate());
    }
}
```

And the Main class:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
public class Main {
    public static void main(String[] args) {
        Date d = new Date();
        d.setYear(115);
        d.setMonth(2);
        d.setDate(23);
        DateProducer producer = new DateProducer();
        producer.setDate(d);
        DatePrinter printer = new DatePrinter();
        MyProgram mp = new MyProgram();
        mp.setProducer(producer);
        mp.setPrinter(printer);
        mp.run();
   }
}
```

I run this program. It works.

The advantage of such modular design is that if I want to use another date producer or another date printer – for instance in order to do tests – there is only one source file that contains the configuration – the *Main* class.

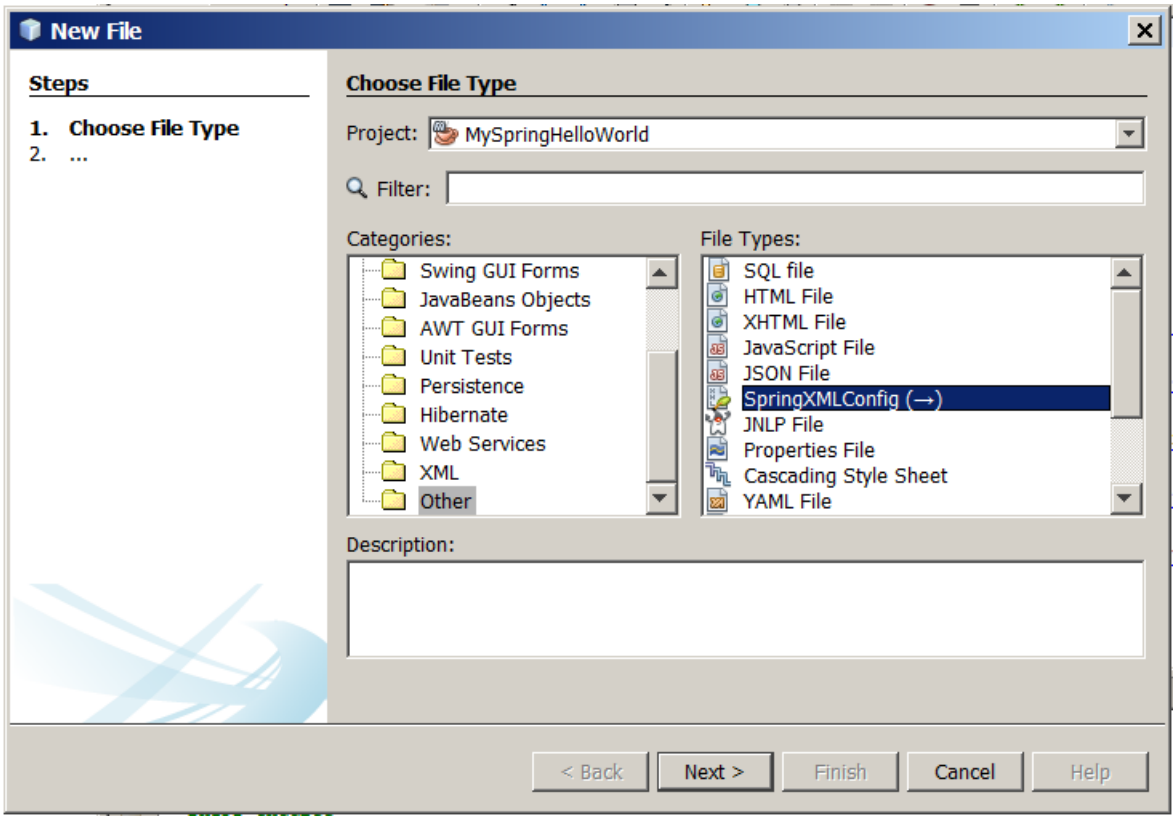
The program will be even more configurable if I move the configuration from the *Main* class to some XML file which will be read during the runtime.

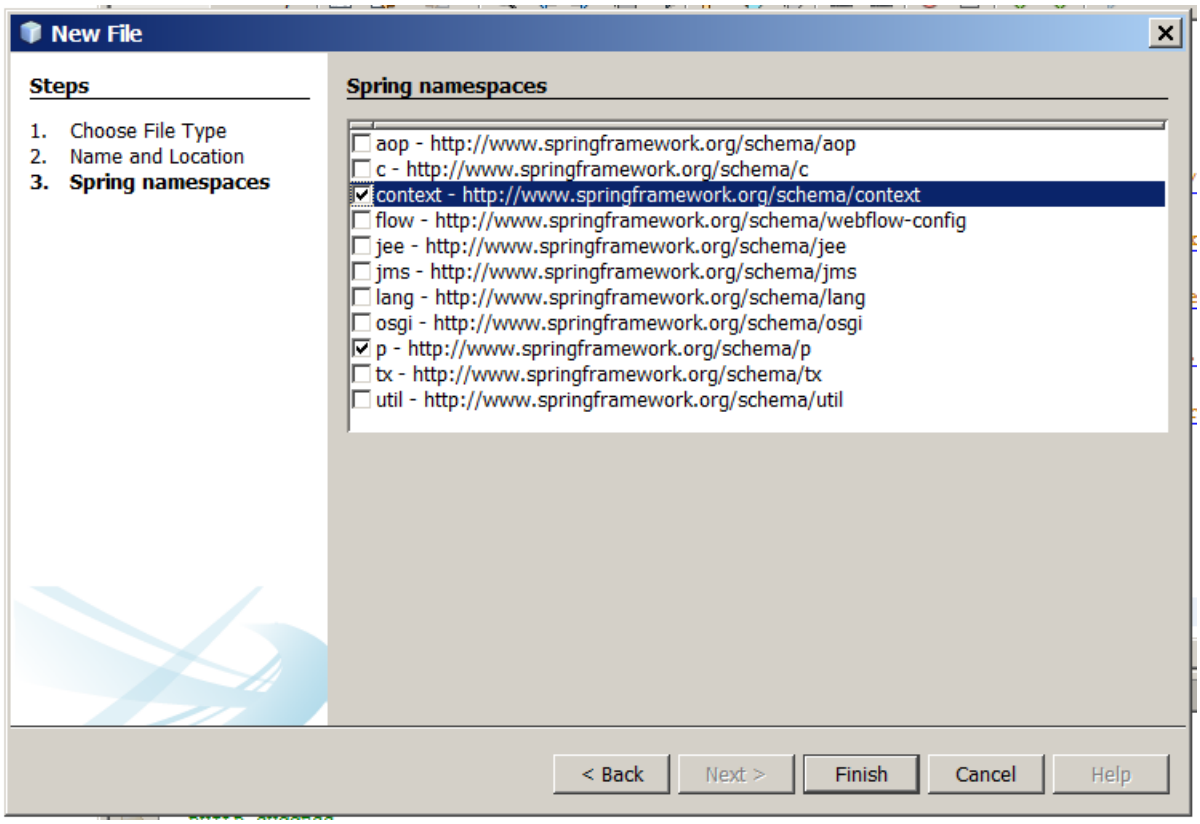
2.1. now I move the configuration to an XML file

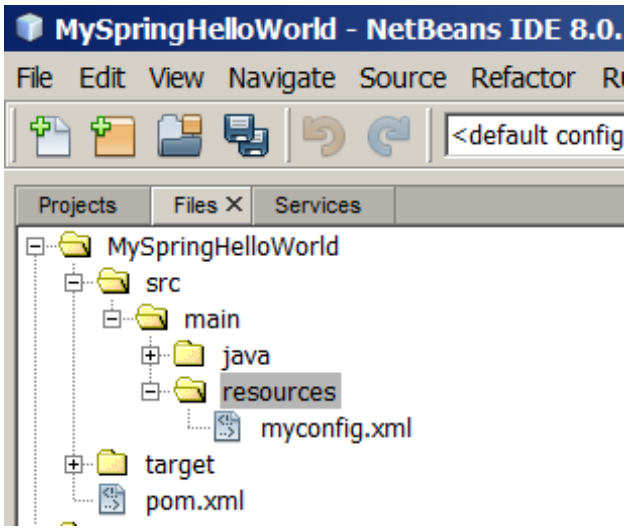
Now I add Spring Framework as a dependency of my project. I add following fragment to the *pom.xml* file of my project:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany</groupId>
    <artifactId>MySpringHelloWorld</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.7</maven.compiler.source>
        <maven.compiler.target>1.7</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.0.3.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

Now in Netbeans I go to *files* tab and there I create file *src/main/resources/myconfig.xml* – I create it as a file of type *Spring XML config*, with *p* and *config* namespaces:







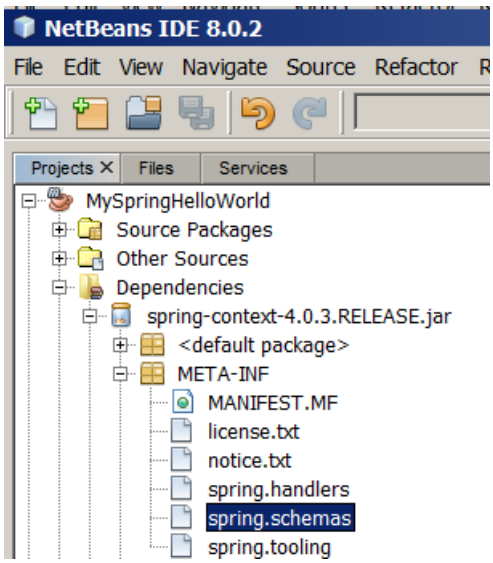
Now I look at this file (before I write anything in it):

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"


xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
</beans>
```

As we can see, the *xsi:schemaLocation* attribute provides URLs of schemas of different namespaces. For instance, we can see that the *context* namespace alias has an URL *http://www.springframework.org/schema/context*, and its XSD is at *http://www.springframework.org/schema/context/spring-context-4.0.xsd*.

During the runtime, Spring will try to download those schemas to validate our XML config file. But, in order not to use Internet connection in vain, Spring's jars contain those XSDs together with a mapping mapping XSDs' URLs to paths of XSD files in jars.

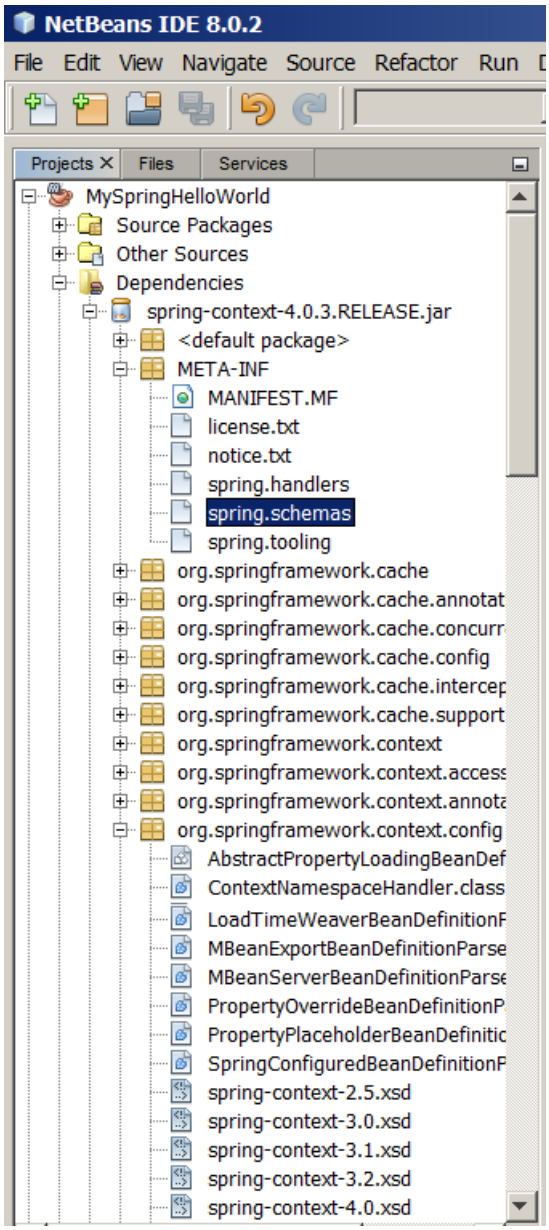
For example, in Netbeans I look at *dependencies -> spring-context-4.0.3.RELEASE.jar -> META-INF -> spring.schemas*:



This is the file with mappings of those XSDs which are in this jar. It looks like that:

```
http\://www.springframework.org/schema/context/spring-context-
2.5.xsd=org/springframework/context/config/spring-context-2.5.xsd
http\://www.springframework.org/schema/context/spring-context-
3.0.xsd=org/springframework/context/config/spring-context-3.0.xsd
http\://www.springframework.org/schema/context/spring-context-
3.1.xsd=org/springframework/context/config/spring-context-3.1.xsd
http\://www.springframework.org/schema/context/spring-context-
3.2.xsd=org/springframework/context/config/spring-context-3.2.xsd
http\://www.springframework.org/schema/context/spring-context-
4.0.xsd=org/springframework/context/config/spring-context-4.0.xsd
(...)
```

As we can see, the mirror of *http://www.springframework.org/schema/context/spring-context-4.0.xsd* is in *org/springframework/context/config/spring-context-4.0.xsd*. I look in this file (it is in the same jar):



It looks like that:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.springframework.org/schema/context"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:beans="http://www.springframework.org/schema/beans"
          xmlns:tool="http://www.springframework.org/schema/tool"
targetNamespace="http://www.springframework.org/schema/context"
          elementFormDefault="qualified"
          attributeFormDefault="unqualified">
     <xsd:import namespace="http://www.springframework.org/schema/
beans"
schemaLocation="http://www.springframework.org/schema/beans/spring
-beans-4.0.xsd"/>
     <xsd:import namespace="http://www.springframework.org/schema/
tool"
schemaLocation="http://www.springframework.org/schema/tool/spring-
tool-4.0.xsd"/>
```

Well, it's just an XSD – nothing special.

Sometimes it's important – for instance because if in our XML configuration file we mention newer versions of XSDs than we have in our jars (for instance because we copied those configs from the web) Spring will actually download them. On stackoverflow there are many questions about problems people have with this behaviour, like:

- https://stackoverflow.com/questions/10680000/you-dont-have-permission-to-access-schema-beans-spring-beans-3-1-xsd-on-this-s
- https://stackoverflow.com/questions/10681864/spring-beans-schema-no-longer-available-on-the-web

Now in this XML configuration file I just created (the one called *myconfig.xml*) I will describe some configurations of some factories. Such named configuration of a factory is called *bean*. Or sometimes this definition is called *bean definition* and the object created by this factory is called *bean.*

I write:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"


xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
</beans>
```

Now in *Main* I comment out the fragment creating Date object and replace it with this:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
public class Main {
    public static void main(String[] args) {
//        Date d = new Date();
//        d.setYear(115);
//        d.setMonth(2);
//        d.setDate(23);
        ApplicationContext context = new
ClassPathXmlApplicationContext("/myconfig.xml");
        Date d = (Date) context.getBean("primaaprilis");
        DateProducer producer = new DateProducer();
        producer.setDate(d);
        DatePrinter printer = new DatePrinter();
        MyProgram mp = new MyProgram();
        mp.setProducer(producer);
        mp.setPrinter(printer);
        mp.run();
   }
}
```

If in the XML configuration file we only have one bean on some class, the method *getBean* can also be given class of the bean – this way we don't need explicit typecast, which gives us type safety:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("/myconfig.xml");
        Date d = context.getBean(Date.class);
        DateProducer producer = new DateProducer();
        producer.setDate(d);
        DatePrinter printer = new DatePrinter();
        MyProgram mp = new MyProgram();
        mp.setProducer(producer);
        mp.setPrinter(printer);
        mp.run();
   }
}
```

We can also provide *getBean* method both bean's name and class:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("/myconfig.xml");
        Date d = context.getBean("primaaprilis", Date.class);
        DateProducer producer = new DateProducer();
        producer.setDate(d);
        DatePrinter printer = new DatePrinter();
        MyProgram mp = new MyProgram();
        mp.setProducer(producer);
        mp.setPrinter(printer);
        mp.run();
   }
}
```

I make sure that the program still works.

Now in the XML configuration file I write the configuration of DateProducer and DatePrinter. This time I have to use *ref* instead of *value* – when we use *value*, the value of value is used to set the property (using editor – see *http://docs.oracle.com/javase/7/docs/api/java/beans/PropertyEditor.html*), when we use *ref*, the property is set to the object whose name is given in ref.

This is what I write:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
    <bean id="myproducer"
class="pl.kuku.myspringhelloworld.DateProducer">
        <property name="date" ref="primaaprilis" />
    </bean>
    <bean id="myprinter"
class="pl.kuku.myspringhelloworld.DatePrinter" />
</beans>
```

Now in Main I write the code which gets DateProducer and DatePrinter from the context.

Now in Main I don't need any more getting the Date from context, so I can comment out this line:

```
Date d = context.getBean("primaaprilis", Date.class);
```

So now *Main* looks like that:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("/myconfig.xml");
        DateProducer producer = context.getBean("myproducer",
DateProducer.class);
        DatePrinter printer = context.getBean("myprinter",
DatePrinter.class);
        MyProgram mp = new MyProgram();
        mp.setProducer(producer);
        mp.setPrinter(printer);
        mp.run();
   }
}
```

I make sure that the program still works.

Now I define *MyProgram* bean in *myconfig.xml*. I use the other syntax to provide values of properties – using attributes from the *p* namespace. I also use the *init-method* property to make Spring run the *run* method on *MyProgram* after creating its instance:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"


xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
    <bean id="myproducer"
class="pl.kuku.myspringhelloworld.DateProducer">
        <property name="date" ref="primaaprilis" />
    </bean>
    <bean id="myprinter"
class="pl.kuku.myspringhelloworld.DatePrinter" />
    <bean id="mymyprogram"
class="pl.kuku.myspringhelloworld.MyProgram"
        init-method="run"
        p:producer-ref="myproducer"
        p:printer-ref="myprinter"
    />
</beans>
```

Now I don't need to get any objects from the context in Main – I only have to create the context. So now Main is very simple:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
public class Main {
    public static void main(String[] args) {
        new ClassPathXmlApplicationContext("/myconfig.xml");
   }
}
```

Now my program has one big advantage – it is possible to configure it without recompiling it. I'll try it. I will reconfigure it to show current date. In *myconfig.xml* I write:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
    <bean id="current_date" class="java.util.Date" />
    <bean id="myproducer"
class="pl.kuku.myspringhelloworld.DateProducer">
        <!-- <property name="date" ref="primaaprilis" /> -->
        <property name="date" ref="current_date" />
    </bean>
    <bean id="myprinter"
class="pl.kuku.myspringhelloworld.DatePrinter" />
    <bean id="mymyprogram"
class="pl.kuku.myspringhelloworld.MyProgram"
        init-method="run"
        p:producer-ref="myproducer"
        p:printer-ref="myprinter"
    />
</beans>
```

I make sure that the program works – it does.

2.2. what if I get the same bean's object several times?

Now I will try to get several times the same bean and I will check whether it is the same object. In Main I write:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("/myconfig.xml");
        Date d1 = (Date) context.getBean("primaaprilis");
        Date d2 = (Date) context.getBean("primaaprilis");
        Date d3 = (Date) context.getBean("primaaprilis");
        System.out.println(String.format("dates' hashcodes: %h,
%h, %h", System.identityHashCode(d1), System.identityHashCode(d2),
System.identityHashCode(d3)));
    }
}
```

The result of this program looks like that:

```
date: Tue Mar 24 10:15:26 CET 2015
dates' hashcodes: d4004b, d4004b, d4004b
```

It proves that when I get several times the same bean, I get the same object.

3. More details on application context configuration

3.1. I create basic Java project using Spring

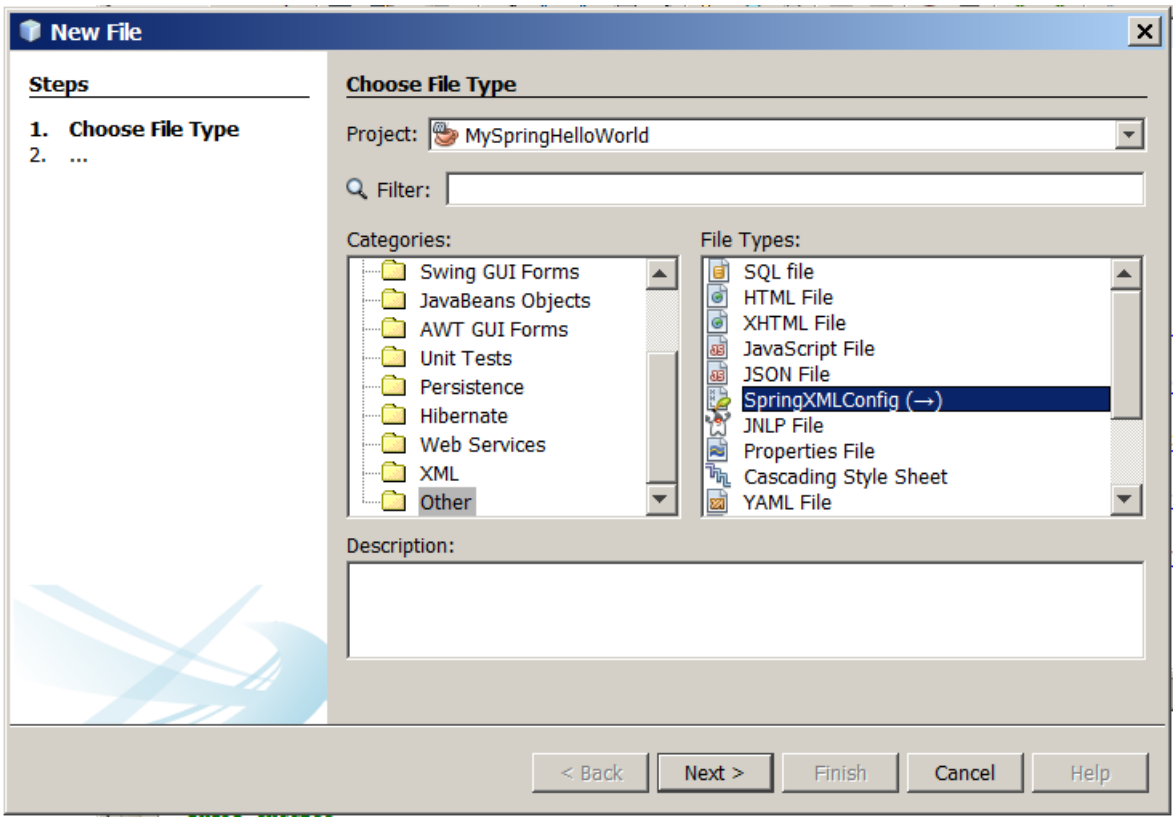
Sometimes I want to do some simple Spring experiments, so I need a simple Netbeans java project using Spring. This chapter describes how to create such project. The project I create in this chapter is a project called *BasicSpringProject*.

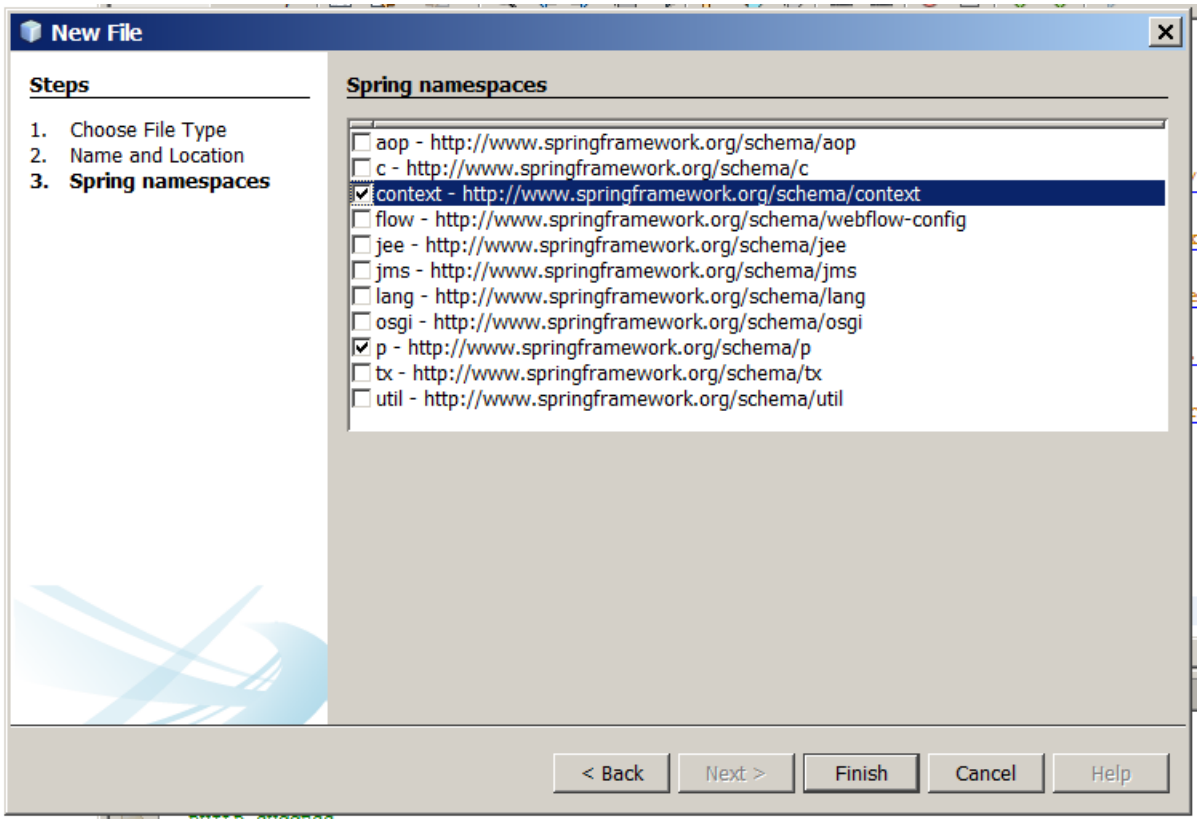
I create in NetBeans a new project of type *maven -> Java application*. I call it *BasicSpringProject*.

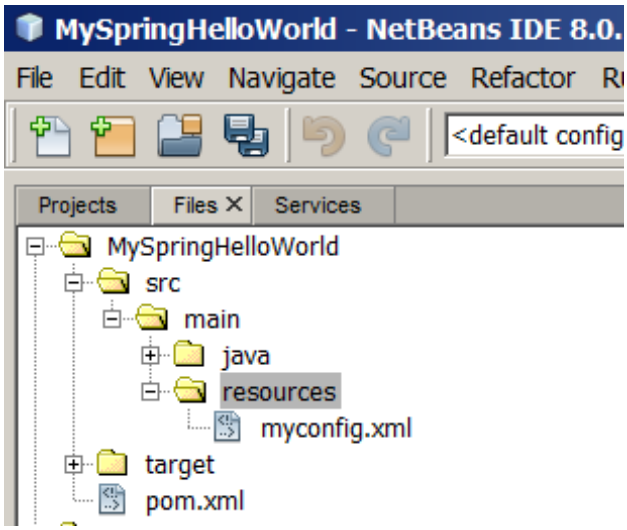
I add Spring Framework as a dependency of my project. I add following fragment to the *pom.xml* file of my project:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany</groupId>
    <artifactId>MySpringHelloWorld</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.7</maven.compiler.source>
        <maven.compiler.target>1.7</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.0.3.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

Now in Netbeans I go to *files* tab and there I create file *src/main/resources/myconfig.xml* – I create it as a file of type *Spring XML config*, with *p* and *config* namespaces (I could also enable more namespaces – maybe all of them? – just in case I need them later):







In this *src/main/resources/myconfig.xml* file I create a simple bean:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
</beans>
```

In the *pl.kuku.basicspringproject* package I create such *MyClass* class:

```
package pl.kuku.basicspringproject;

import java.util.Date;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;

public class MyClass {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("/myconfig.xml");
        Date d = (Date) context.getBean("primaaprilis");
        System.out.println("date: " + d);
    }
}
```

I run the project. It works and prints out something like:

```
date: Wed Apr 01 17:41:12 CEST 2015
```

And this is the basic Netbeans project using Spring. I will use it in some later chapters.

3.2. how to set properties of different types

The program I write in this chapter and in following chapters is in the project called *SpringSettingProperties*.

3.2.1. informal, inclomplete and simplified but useful grammar of the configuration of an application context

There are several ways to define an object in the configuration of an application context: using a *bean*, *value, null, ref*, *idref*, *array*, *map*, *list*, *set* or *props* element.

Of those, only *bean* elements can have a name, so on the top level of the configuration of an application context we put only *bean* elements.

Inside a *bean* we can put *property* elements. A *property* element must have a *name* attribute and must contain some object definition (it is, a *bean*, *value, null, ref*, *idref*, *array*, *map*, *list*, *set* or *props* element).

An *array* and a *list* work the same – inside them I put any number of object definitions.

A *map* element may contain any number of *entry* elements. Each such *entry* element produces one key-value pair. Each such *entry* element must contain one *key* element (which produces the key of this key-value pair) and one object definition (which produces the value of this key-value pair). The *key* element must contain one object definition.

There are many shortcut ways to write things – often we can use attributes instead of children elements.

3.2.2. examples

I do a copy of *BasicSpringProject* project. I call it *SpringSettingProperties*.

I create is this project such class:

```
package pl.kuku.basicspringproject;

public class Person {
    private String name;
    private int age;
}
```

I generate in it getters, setters and toString:

```
package pl.kuku.basicspringproject;

public class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{" + "name=" + name + ", age=" + age + '}';
    }

}
```

I also create such class:

```
package pl.kuku.basicspringproject;

import java.util.List;
import java.util.Map;

public class JustATest {
    private int someNumber;
```

```
    private Person somePerson;
    private Person anotherPerson;
    private int[] arrayOfInts;
    private List<String> listOfStrings;
    private List<Person> listOfPersons;
    private Map<String, String> mapOfStringsAndStrings;
    private Map<String, Person> mapOfStringAndPersons;
    private Map<String, List<Person>>
mapOfStringsAndListsOfPersons;

    public int getSomeNumber() {
        return someNumber;
    }

    public void setSomeNumber(int someNumber) {
        this.someNumber = someNumber;
    }

    public Person getSomePerson() {
        return somePerson;
    }

    public void setSomePerson(Person somePerson) {
        this.somePerson = somePerson;
    }

    public Person getAnotherPerson() {
        return anotherPerson;
    }

    public void setAnotherPerson(Person anotherPerson) {
        this.anotherPerson = anotherPerson;
    }

    public int[] getArrayOfInts() {
        return arrayOfInts;
    }

    public void setArrayOfInts(int[] arrayOfInts) {
        this.arrayOfInts = arrayOfInts;
    }

    public List<String> getListOfStrings() {
        return listOfStrings;
    }

    public void setListOfStrings(List<String> listOfStrings) {
        this.listOfStrings = listOfStrings;
    }
```

```
    public List<Person> getListOfPersons() {
        return listOfPersons;
    }

    public void setListOfPersons(List<Person> listOfPersons) {
        this.listOfPersons = listOfPersons;
    }

    public Map<String, String> getMapOfStringsAndStrings() {
        return mapOfStringsAndStrings;
    }

    public void setMapOfStringsAndStrings(Map<String, String>
mapOfStringsAndStrings) {
        this.mapOfStringsAndStrings = mapOfStringsAndStrings;
    }

    public Map<String, Person> getMapOfStringAndPersons() {
        return mapOfStringAndPersons;
    }

    public void setMapOfStringAndPersons(Map<String, Person>
mapOfStringAndPersons) {
        this.mapOfStringAndPersons = mapOfStringAndPersons;
    }

    public Map<String, List<Person>>
getMapOfStringsAndListsOfPersons() {
        return mapOfStringsAndListsOfPersons;
    }

    public void setMapOfStringsAndListsOfPersons(Map<String,
List<Person>> mapOfStringsAndListsOfPersons) {
        this.mapOfStringsAndListsOfPersons =
mapOfStringsAndListsOfPersons;
    }

    @Override
    public String toString() {
        String ints = "";
        if (arrayOfInts == null) {
            ints = "null";
        } else {
            for (int i : arrayOfInts) {
                ints += i + " ";
            }
        }
```

```
        return "JustATest{" + "someNumber=" + someNumber + ",
somePerson=" + somePerson + ", anotherPerson=" + anotherPerson +
", arrayOfInts=" + ints + ", listOfStrings=" + listOfStrings + ",
listOfPersons=" + listOfPersons + ", mapOfStringsAndStrings=" +
mapOfStringsAndStrings + ", mapOfStringAndPersons=" +
mapOfStringAndPersons + ", mapOfStringsAndListsOfPersons=" +
mapOfStringsAndListsOfPersons + '}';
    }

}
```

Most of this class is generated by Netbeans. I only wrote private attributes and altered the *toString* method so it prints out all numbers of the *arrayOfInts*.

Now I will create a bean of type JustATest and I will set all those properties.

In *MyClass* class I get a bean named *parrot* (I haven't defined it yet in *myconfig.xml*, but I will do it very soon) from the application context and I print it:

```
package pl.kuku.basicspringproject;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyClass {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("/myconfig.xml");
        JustATest t = (JustATest) context.getBean("parrot");
        System.out.println("the bean is: " + t);
    }
}
```

So now I can try (in *myconfig.xml*) different ways of defining this *parrot* bean and just run my program to check if it works.

First I try (in *myconfig.xml*) to set the *someNumber* property. I can do it this way:

```
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
     <property name="someNumber">
          <value>57005</value>
     </property>
</bean>
```

Or this way:

```
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
     <property name="someNumber" value="57005" />
</bean>
```

Or this way:

```
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest"
p:someNumber="57005" />
```

In Spring there are many ways to define properties. Here I will not show all of them.

Then I try (in *myconfig.xml*) to set the *somePerson* property. I can't do this using *value* element. I can do it this way, by putting element *bean* in the place where it is needed:

```
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
<property name="somePerson">
     <bean class="pl.kuku.basicspringproject.Person"
p:name="Piotr" p:age="40" />
</property>
</bean>
```

Or I can define it elsewhere and define refer to it using element *ref:*

```
<bean name="p1" class="pl.kuku.basicspringproject.Person"
p:name="Piotr" p:age="40" />
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
     <property name="somePerson">
          <ref bean="p1" />
     </property>
</bean>
```

There is a shortcut for it:

```
<bean name="p1" class="pl.kuku.basicspringproject.Person"
p:name="Piotr" p:age="40" />
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
     <property name="somePerson" ref="p1" />
</bean>
```

As we can see, when I need somewhere a bean, I can put element *bean* in place, or I can refer to some bean defined elsewhere using a *ref* element.

As we can see, when I write the configuration file of an application context and I need to produce an object, I can use *value*, *bean* or *ref.*

Now I try (in *myconfig.xml*) to set the *arrayOfInts* property. I can do it with an *array* element into which I put many *value* elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean name="parrot"
class="pl.kuku.basicspringproject.JustATest">
        <property name="arrayOfInts">
            <array>
                <value>3</value>
                <value>5</value>
                <value>7</value>
            </array>
        </property>
    </bean>
</beans>
```

(now I run my program and I can see that it works).

I can also not use an *array* element but a *list* element:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean name="parrot"
class="pl.kuku.basicspringproject.JustATest">
        <property name="arrayOfInts">
            <list>
                <value>3</value>
                <value>5</value>
                <value>7</value>
            </list>
        </property>
    </bean>
</beans>
```

(now I run my program and I can see that it works).

When I read the official Spring documentation, I could not find there anything about the *array* element. However, I can find an XSD file for the *http://www.springframework.org/schema/beans* namespace – it is in *spring-beans-4.0.3.RELEASE.jar*, in the *org.springframework.beans.factory.xml* namespace, in *spring-beans-4.0.xsd* file. In this XSD file I can find that an element *property* is of type *propertyType*:

```
<xsd:element name="property" type="propertyType">
  <xsd:annotation>
    <xsd:documentation><![CDATA[
Bean definitions can have zero or more properties.
Property elements correspond to JavaBean setter methods exposed
by the bean classes. Spring supports primitives, references to
other
beans in the same or related factories, lists, maps and
properties.
    ]]></xsd:documentation>
  </xsd:annotation>
</xsd:element>
```

And in the definition of this type I read:

```
<xsd:complexType name="propertyType">
  <xsd:sequence>
    <xsd:element ref="description" minOccurs="0"/>
    <xsd:choice minOccurs="0" maxOccurs="1">
      <xsd:element ref="meta"/>
      <xsd:element ref="bean"/>
      <xsd:element ref="ref"/>
      <xsd:element ref="idref"/>
      <xsd:element ref="value"/>
      <xsd:element ref="null"/>
      <xsd:element ref="array"/>
      <xsd:element ref="list"/>
      <xsd:element ref="set"/>
      <xsd:element ref="map"/>
      <xsd:element ref="props"/>
      <xsd:any namespace="##other" processContents="strict"/>
    </xsd:choice>
  </xsd:sequence>
```

And the definition of the *array* element says:

```
<xsd:element name="array">
  <xsd:annotation>
    <xsd:documentation><![CDATA[
An array can contain multiple inner bean, ref, collection, or
value elements.
This configuration element will always result in an array, even
when being
defined e.g. as a value for a map with value type Object.
    ]]></xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="listOrSetType">
        <xsd:attribute name="merge" default="default"
type="defaultable-boolean">
          <xsd:annotation>
            <xsd:documentation><![CDATA[
Enables/disables merging for collections when using parent/child
beans.
            ]]></xsd:documentation>
          </xsd:annotation>
        </xsd:attribute>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

And in the definition of the *list* element I read:

```
<xsd:element name="list">
  <xsd:annotation>
    <xsd:documentation><![CDATA[
A list can contain multiple inner bean, ref, collection, or value
elements.
A list can also map to an array type; the necessary conversion is
performed automatically.
    ]]></xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="listOrSetType">
        <xsd:attribute name="merge" default="default"
type="defaultable-boolean">
          <xsd:annotation>
            <xsd:documentation><![CDATA[
Enables/disables merging for collections when using parent/child
beans.
            ]]></xsd:documentation>
          </xsd:annotation>
        </xsd:attribute>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

This explains a lot. The syntax of *array* and *list* is the same. The documentation inlined in XSD says that we can use the *list* element to initialize arrays.

As we can see, even though XSD files generally define only syntax and not semantics, they can be useful.

Now I set the *listOfStrings* property with a *list* element (again, with many *value* elements in it):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean name="parrot"
class="pl.kuku.basicspringproject.JustATest">
        <property name="listOfStrings">
            <list>
                <value>Jalisco</value>
                <value>Chihuahua</value>
                <value>Durango</value>
            </list>
        </property>
    </bean>
</beans>
```

However, I can also initialize a list with an *array* element:

```
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
     <property name="listOfStrings">
          <array>
               <value>Jalisco</value>
               <value>Chihuahua</value>
               <value>Durango</value>
          </array>
     </property>
</bean>
```

Now I will set the *listOfPersons* property. Of course, I can't do it using *value* elements:

```
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
     <property name="listOfPersons">
          <list>
               <value>Jalisco</value>
               <value>Chihuahua</value>
               <value>Durango</value>
          </list>
     </property>
</bean>
```

This will produce such error:

```
(...) Cannot convert value of type [java.lang.String] to required
type [pl.kuku.basicspringproject.Person] for property
'listOfPersons[0]': no matching editors or conversion strategy
found (...)
```

As usual with types that are not convertable from Strings, I can define persons outside of a *parrot* bean and then use *ref* elements:

```
<bean name="person1" class="pl.kuku.basicspringproject.Person"
p:name="Piotr" p:age="40" />
<bean name="person2" class="pl.kuku.basicspringproject.Person"
p:name="Zosia" p:age="6" />
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
     <property name="listOfPersons">
          <list>
               <ref bean="person1" />
               <ref bean="person2" />
          </list>
     </property>
</bean>
```

I can also use nested *bean* elements:

```
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
     <property name="listOfPersons">
          <list>
               <bean class="pl.kuku.basicspringproject.Person"
p:name="Piotr" p:age="40" />
               <bean class="pl.kuku.basicspringproject.Person"
p:name="Zosia" p:age="6" />
          </list>
     </property>
</bean>
```

Now I will set the *mapOfStringsAndStrings* property. I can do it this way:

```
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
    <property name="mapOfStringsAndStrings">
        <map>
            <entry>
                <key><value>red</value></key>
                <value>vermelho</value>
            </entry>
        </map>
    </property>
</bean>
```

or this way:

```
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
    <property name="mapOfStringsAndStrings">
        <map>
            <entry key="red">
                <value>vermelho</value>
            </entry>
        </map>
    </property>
</bean>
```

Then I set the *mapOfStringAndPersons* property. I can do it for instance this way:

```
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
    <property name="mapOfStringAndPersons">
        <map>
            <entry key="red">
                <bean class="pl.kuku.basicspringproject.Person">
                    <property name="age">
                        <value>40</value>
                    </property>
                    <property name="name">
                        <value>Piotr</value>
                    </property>
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

Then I set the *mapOfStringsAndListsOfPersons* property. I can do it for instance this way:

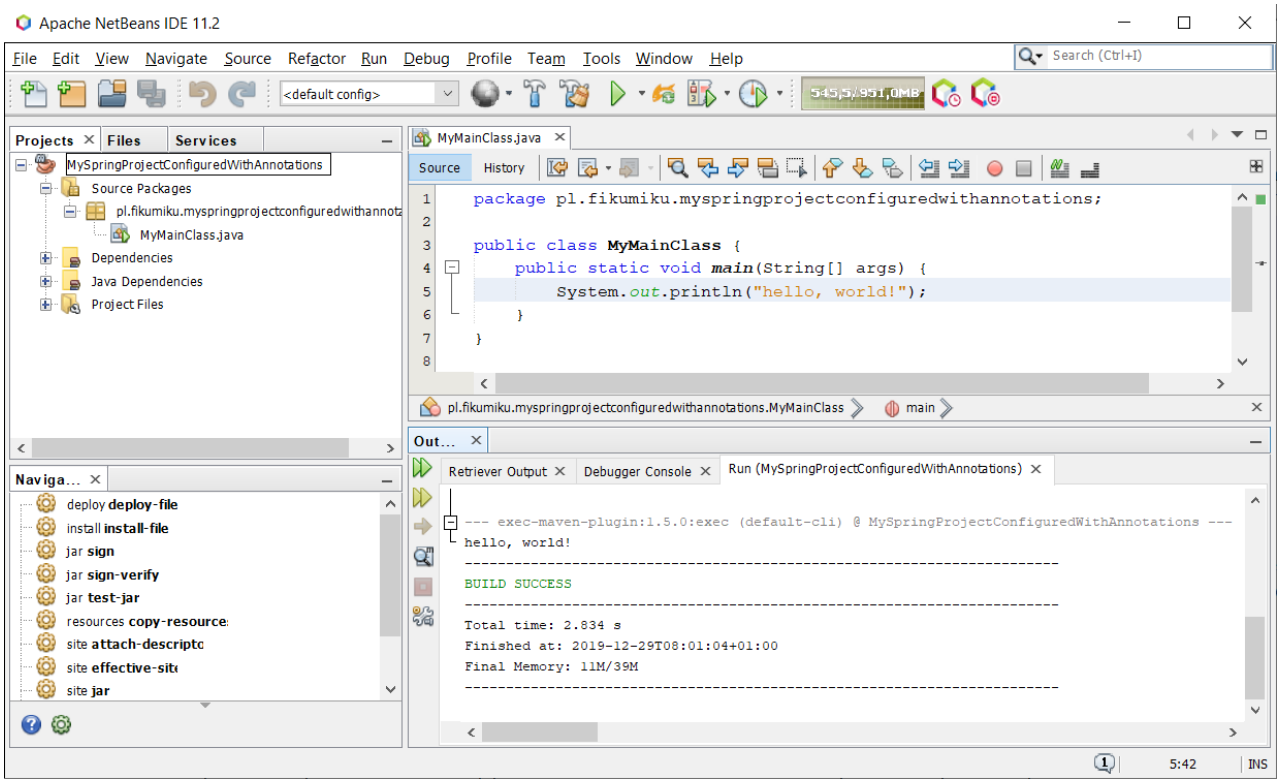
```
<bean name="parrot" class="pl.kuku.basicspringproject.JustATest">
    <property name="mapOfStringsAndListsOfPersons">
        <map>
            <entry key="red">
                <list>
                    <bean
class="pl.kuku.basicspringproject.Person">
                        <property name="age">
                            <value>40</value>
                        </property>
                        <property name="name">
                            <value>Piotr</value>
                        </property>
                    </bean>
                    <bean
class="pl.kuku.basicspringproject.Person" p:name="Zosia" p:age="6"
/>
                </list>
            </entry>
        </map>
    </property>
</bean>
```

4. Using Java annotations – instead of XML – to configure XML

So far we were configuring context with XML files. There is also another way of configuring contexts. Some contexts can be configured by putting into them a bean with special annotations. In this chapter I will create a project configured this way.

4.1. basic example

In Netbeans I create a new project of type java with *maven → java application*, I name it *MySpringProjectConfiguredWithAnnotations*. I create in it a class, I call it *MyMainClass*. In this class I create a main method which prints *hello, world!*. I run the project, it works.



In pom.xml of the project I add a dependency to *spring-context*:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>pl.fikumiku</groupId>

<artifactId>MySpringProjectConfiguredWithAnnotations</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>9</maven.compiler.source>
        <maven.compiler.target>9</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.2.2.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

I create two classes which will be used as Spring beans.

MyBean:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;
public class MyBean {
}
```

MySecondBean:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;
public class MySecondBean {
}
```

In the main method of *MyMainClass* I create an instance of *AnnotationConfigApplicationContext* (this class is one of those contexts which can be configured with annotations). I register my beans in it, then I try to get them from the context.

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import
org.springframework.context.annotation.AnnotationConfigApplication
Context;

public class MyMainClass {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();
        ctx.register(MyBean.class, MySecondBean.class);
        System.out.println("first: " + ctx.getBean(MyBean.class));
        System.out.println("second: " +
ctx.getBean(MySecondBean.class));
    }
}
```

I run the project. It fails:

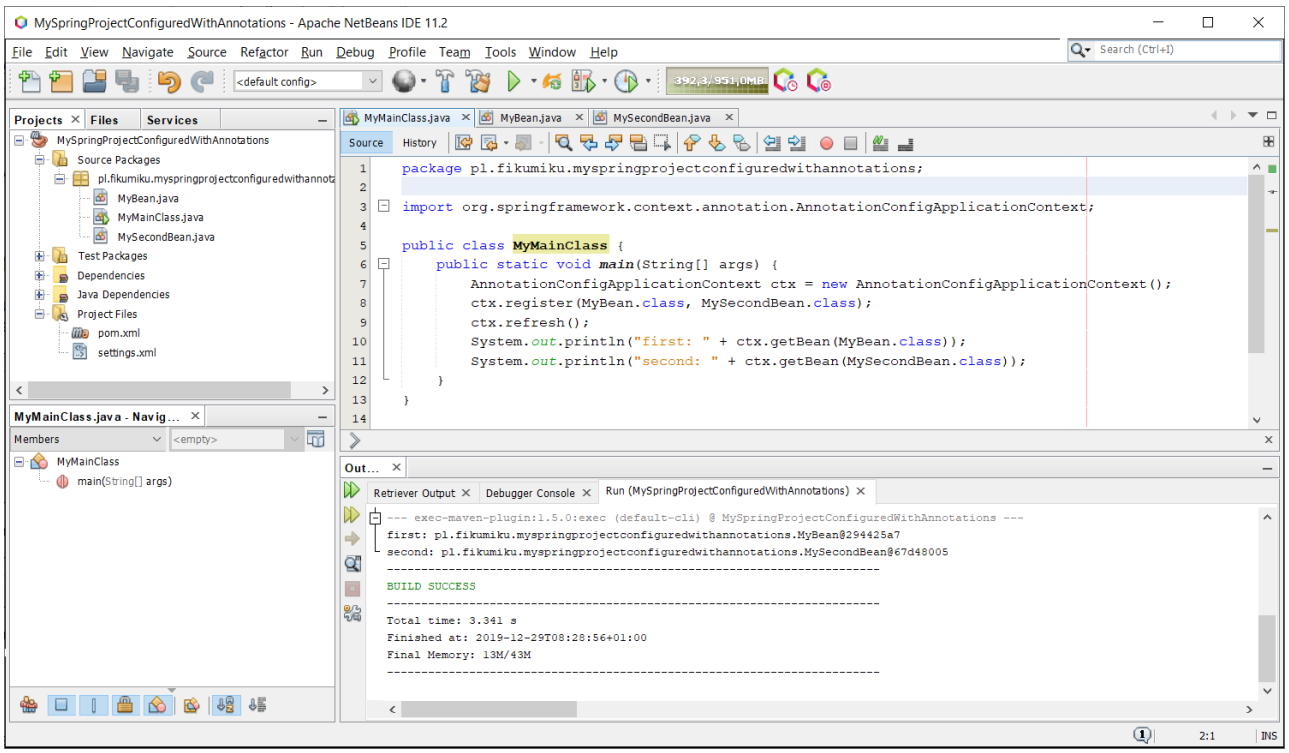
```
Exception in thread "main" java.lang.IllegalStateException:
org.springframework.context.annotation.AnnotationConfigApplication
Context@5abca1e0 has not been refreshed yet
    at
org.springframework.context.support.AbstractApplicationContext.ass
ertBeanFactoryActive(AbstractApplicationContext.java:1095)
    at
org.springframework.context.support.AbstractApplicationContext.get
Bean(AbstractApplicationContext.java:1125)
    at
pl.fikumiku.myspringprojectconfiguredwithannotations.MyMainClass.m
ain(MyMainClass.java:9)
```

It has failed because after new beans are registered in a context, the context must be refreshed. I change *MyMainClass* so it refreshes the context:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import
org.springframework.context.annotation.AnnotationConfigApplication
Context;

public class MyMainClass {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();
        ctx.register(MyBean.class, MySecondBean.class);
        ctx.refresh();
        System.out.println("first: " + ctx.getBean(MyBean.class));
        System.out.println("second: " +
ctx.getBean(MySecondBean.class));
    }
}
```

I run it. Now it works:

```
first:
pl.fikumiku.myspringprojectconfiguredwithannotations.MyBean@294425
a7
second:
pl.fikumiku.myspringprojectconfiguredwithannotations.MySecondBean@
67d48005
```



Now I will configure the context with an annotated bean. This annotated bean will be analogue of an XML file and its annotations will be analogues of XML tags.

I create a class called *MyConfiguration*. In this class I create methods which return objects of classes *MyBean* and *MySecondBean*. I annotate these methods with *@Bean*. These methods will work like a *<bean>* tag in an XML configuration:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;
import org.springframework.context.annotation.Bean;
public class MyConfiguration {
    @Bean
    public MyBean mb() {
        return new MyBean();
    }
    @Bean
    public MySecondBean msb() {
        return new MySecondBean();
    }
}
```

I change *MyMainClass* so it registers only one bean – *MyConfiguration* – in the context:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import
org.springframework.context.annotation.AnnotationConfigApplication
Context;

public class MyMainClass {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();
//        ctx.register(MyBean.class, MySecondBean.class);
        ctx.register(MyConfiguration.class);
        ctx.refresh();
        System.out.println("first: " + ctx.getBean(MyBean.class));
        System.out.println("second: " +
ctx.getBean(MySecondBean.class));
    }
}
```

I run the project. It still works:

```
first:
pl.fikumiku.myspringprojectconfiguredwithannotations.MyBean@1fa121
e2
second:
pl.fikumiku.myspringprojectconfiguredwithannotations.MySecondBean@
7eac9008
```

The context registered *MyConfiguration*, it found *@Bean* annotations and registered two beans – *MyBean* and *MySecondBean* – according to these *@Bean* annotations.

4.2. how will it work with package scanning

Now I do an experiment. I change *MyMainClass* so it does not directly register *MyConfiguration* bean: instead I make the context scan a Java package and I see if it will work. Here are my changes to *MyMainClass*:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import
org.springframework.context.annotation.AnnotationConfigApplication
Context;

public class MyMainClass {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();
//        ctx.register(MyConfiguration.class);

ctx.scan("pl.fikumiku.myspringprojectconfiguredwithannotations");
        ctx.refresh();
        System.out.println("first: " + ctx.getBean(MyBean.class));
        System.out.println("second: " +
ctx.getBean(MySecondBean.class));
    }
}
```

I run the project. It fails:

```
Exception in thread "main"
org.springframework.beans.factory.NoSuchBeanDefinitionException:
No qualifying bean of type
'pl.fikumiku.myspringprojectconfiguredwithannotations.MyBean'
available
     at
org.springframework.beans.factory.support.DefaultListableBeanFacto
ry.getBean(DefaultListableBeanFactory.java:351)
     at
org.springframework.beans.factory.support.DefaultListableBeanFacto
ry.getBean(DefaultListableBeanFactory.java:342)
     at
org.springframework.context.support.AbstractApplicationContext.get
Bean(AbstractApplicationContext.java:1126)
     at
pl.fikumiku.myspringprojectconfiguredwithannotations.MyMainClass.m
ain(MyMainClass.java:12)
```

It fails because when the context scans a package, it does not use all classes found in the package – it only uses classes annotated with *@Component* or classes annotated with annotation annotated with *@Component*. So I annotate *MyConfiguration* with *@Component*:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;

@Component
public class MyConfiguration {
    @Bean
    public MyBean mb() {
        return new MyBean();
    }
    @Bean
    public MySecondBean msb() {
        return new MySecondBean();
    }
}
```

I run the project. Now it works:

```
first:
pl.fikumiku.myspringprojectconfiguredwithannotations.MyBean@38e79a
e3
second:
pl.fikumiku.myspringprojectconfiguredwithannotations.MySecondBean@
63070bab
```

However, usually we annotate a configuration class not with generic *@Component* annotation but with *@Configuration* annotation (note that *@Configuration* is annotated with *@Component*). I can change *MyConfiguration* to:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyConfiguration {
    @Bean
    public MyBean mb() {
        return new MyBean();
    }
    @Bean
    public MySecondBean msb() {
        return new MySecondBean();
    }
}
```

When I run the project, it works the same as before.

4.3. why should we annotate configuration class with *@Configuration* rather with *@Component*

From what we have seen so far we could think that – at least as far as we talk about behavior – it doesn't make any difference if we if annotate a configuration class with *@Configuration*, *@Component* or any other annotation annotated with *@Component* (e.g. *@Controller*). It is not true. There are corner case differences between behavior of *@Configuration* and other *@Component* annotatations, so we should not be smart, we should just always annotate configuration classes with *@Configuration*. For instance, let's see one difference between *@Configuration* and *@Component*. I change *MyConfiguration* so it is annotated with *@Component*:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;

@Component
public class MyConfiguration {
    @Bean
    public MyBean mb() {
        return new MyBean();
    }
    @Bean
    public MySecondBean msb() {
        return new MySecondBean();
    }
}
```

Then I change *MyMainClass* so it gets from component *MyConfiguration*, then calls on this object *mb()* method two times – receiving two beans; and finally it prints all three objects:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import
org.springframework.context.annotation.AnnotationConfigApplication
Context;

public class MyMainClass {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();

ctx.scan("pl.fikumiku.myspringprojectconfiguredwithannotations");
        ctx.refresh();
        System.out.println("first: " + ctx.getBean(MyBean.class));
        System.out.println("second: " +
ctx.getBean(MySecondBean.class));
        MyConfiguration mc = ctx.getBean(MyConfiguration.class);
        System.out.println("my configuration bean: " + mc);
        MyBean mb1 = mc.mb();
        MyBean mb2 = mc.mb();
        System.out.println("MyBean bean, two times: " + mb1 + " -
" + mb2);
    }
}
```

I run the project. The result is quite obvious – *MyConfiguration* bean is an instance of *MyConfiguration* class and each invocation of *mb()* method has returned a new instance of *MyBean*:

```
first:
pl.fikumiku.myspringprojectconfiguredwithannotations.MyBean@38e79a
e3
second:
pl.fikumiku.myspringprojectconfiguredwithannotations.MySecondBean@
63070bab
my configuration bean:
pl.fikumiku.myspringprojectconfiguredwithannotations.MyConfigurati
on@68e5eea7
MyBean bean, two times:
pl.fikumiku.myspringprojectconfiguredwithannotations.MyBean@21b2e7
68 -
pl.fikumiku.myspringprojectconfiguredwithannotations.MyBean@6a03bc
b1
```

Now I change *MyConfiguration* class – I annotate it with *@Configuration* instead of *@Component*:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyConfiguration {
    @Bean
    public MyBean mb() {
        return new MyBean();
    }
    @Bean
    public MySecondBean msb() {
        return new MySecondBean();
    }
}
```

I run the project again and now I see that magic is happening. This time *MyConfiguration* bean is an instance of a strange class called *MyConfiguration$$EnhancerBySpringCGLIB$$69178ab0* and each invocation of *mb()* method has returned the same instance of *MyBean*:

```
first:
pl.fikumiku.myspringprojectconfiguredwithannotations.MyBean@534a5a98
second:
pl.fikumiku.myspringprojectconfiguredwithannotations.MySecondBean@4f80542f
my configuration bean:
pl.fikumiku.myspringprojectconfiguredwithannotations.MyConfiguration$$EnhancerBySpringCGLIB$$69178ab0@60bd273d
MyBean bean, two times:
pl.fikumiku.myspringprojectconfiguredwithannotations.MyBean@534a5a98 -
pl.fikumiku.myspringprojectconfiguredwithannotations.MyBean@534a5a98
```

It is because this time context has created a proxy object for *MyConfiguration* bean and this proxy object caches the result of *mb()*.

This is a good behaviour. In Spring we usually want that each bean is a singleton[1] – and this behavior makes each bean a singleton when, for instance, two *@Bean* annotated methods call another (third) *@Bean* annotated method. Spring documentation (see: *https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans-java*) says:

1 It can be more complicated when we play with scopes: https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans-factory-scopes

Annotating a class with @Configuration indicates that its primary purpose is as a source of bean definitions. Furthermore, @Configuration classes let inter-bean dependencies be defined by calling other @Bean methods in the same class.

For instance, I create a class called *Logger*:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

public class Logger {
    private final FileWriter fw;

    public Logger() {
        try {
            fw = new FileWriter("log-" + new Random().nextInt() +
".txt");
        } catch (IOException ex) {
            throw new RuntimeException(ex);
        }
    }

    public void log(String message) {
        try {
            fw.write(message + "\n");
            fw.flush();
        } catch (IOException ex) {
            throw new RuntimeException(ex);
        }
    }
}
```

This class creates a log file and can log to this file (FIXME: btw, what should I do when I want to close this file when my program exits? In this example I had to do flush() after each write, otherwise the log file was empty, but I would like to make it in some more elegant way. See: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Bean.html#destroyMethod--).

I also create two simple classes which use the logger – *Counter* and *ReverseCounter*. *Counter* class is:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

public class Counter {
    private int cnt;
    private final Logger logger;

    public Counter(Logger logger) {
        this.logger = logger;
    }

    public int increase() {
        String message = "counter increased from " + cnt;
        cnt++;
        message += " to " + cnt;
        logger.log(message);
        return cnt;
    }
}
```

ReverseCounter class is:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

public class ReverseCounter {
    private int cnt;
    private final Logger logger;

    public ReverseCounter(Logger logger) {
        this.logger = logger;
    }

    public int decrease() {
        String message = "counter decreased from " + cnt;
        cnt--;
        message += " to " + cnt;
        logger.log(message);
        return cnt;
    }
}
```

I annotate my configuration class with *@Configuration* and I register three new beans in the configuration class:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;

//@Component
@Configuration
public class MyConfiguration {
    @Bean
    public Logger logger() {
        return new Logger();
    }
    @Bean
    public Counter counter() {
        return new Counter(logger());
    }
    @Bean
    public ReverseCounter reverseCounter() {
        return new ReverseCounter(logger());
    }
}
```

I run the project. I see that it has created one log file which contents is:

```
counter increased from 0 to 1
counter increased from 1 to 2
counter decreased from 0 to -1
counter decreased from -1 to -2
counter decreased from -2 to -3
```

This was a good behaviour.

Now I change my configuration class – I annotate it with *@Component* instead of *@Configuration*:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;

@Component
//@Configuration
public class MyConfiguration {
    @Bean
    public Logger logger() {
        return new Logger();
    }
    @Bean
    public Counter counter() {
        return new Counter(logger());
    }
    @Bean
    public ReverseCounter reverseCounter() {
        return new ReverseCounter(logger());
    }
}
```

Now I run my project again. This time it has created three log files (because it has created three instances of *Logger*: one object is a bean, another object belongs to *Counter*, the other one belongs to *ReverseCounter*). One log file is empty, another log file has such contents:

```
counter increased from 0 to 1
counter increased from 1 to 2
```

the other one has such contents:

```
counter decreased from 0 to -1
counter decreased from -1 to -2
counter decreased from -2 to -3
```

For more information about why should we rather use *@Configuration* (and not *@Component*) annotation, see: *https://stackoverflow.com/questions/59518342/behaviour-difference-between-configuration-and-component*

4.4. we can provide configuration class to context constructor

So far we were providing a configuration class to the context either with *register()* or with *scan()* method. There is another, quite popular way. The constructor of *AnnotationConfigApplicationContext* accepts a varargs array of classes which will be registered as beans. It is quite common to provide in this parameter a configuration class. Now I will change the project so it works this way.

I make sure that *MyConfiguration* is annotated with *@Configuration*:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;

//@Component
@Configuration
public class MyConfiguration {
    @Bean
    public Logger logger() {
        return new Logger();
    }
    @Bean
    public Counter counter() {
        return new Counter(logger());
    }
    @Bean
    public ReverseCounter reverseCounter() {
        return new ReverseCounter(logger());
    }
}
```

I change *MyMainClass* so it passes the configuration class to the context's constructor:

```
package pl.fikumiku.myspringprojectconfiguredwithannotations;

import
org.springframework.context.annotation.AnnotationConfigApplication
Context;

public class MyMainClass {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(MyConfiguration.class);
//
ctx.scan("pl.fikumiku.myspringprojectconfiguredwithannotations");
//        ctx.refresh();
        Counter cnt = ctx.getBean(Counter.class);
```

```
        ReverseCounter rcnt = ctx.getBean(ReverseCounter.class);
        cnt.increase();
        cnt.increase();
        rcnt.decrease();
        rcnt.decrease();
        rcnt.decrease();
    }
}
```

I run the project. It works – it creates a correct log file.

4.5. why do I need configuration at all

You may wonder why do we need a configuration class at all. Couldn't we just annotate our beans with *@Component* and make component scan a package, so it just discovers all beans? As far as I understand, there are (at least) two reasons. The first reason is that sometimes I may want to have two beans of the same class. I can do it with XML configuration (I just need two *<bean>* tags in it), I can do it with configuration class (I just need two *@Bean* annotated methods in it), but I can't do it without configuration class, using only package scanning and *@Component* annotation on beans. The other reason is that if I have in my XML configuration file any tag other than *<bean>* (for instance, if I have *<context:component-scan>* tag) and I want to get rid of the XML configuration then I can do it with configuration class – because each component tag has an annotation which do the same (for instance, *@ComponentScan* annotation does the same as *<context:component-scan>* tag), but I can't do it with just package scanning and *@Component* annotation on beans.

4.6. more info

For more information about configuring Spring with annotations, see: *https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans-java* . It is a good description, it is really worth reading.

5. Using a DataSource

The program I write in this chapter and in following chapters is in the project *MySpringHelloWorld* – the same project I was using several chapters before.

5.1. I create and use a DataSource

Now I will create a DataSource, and then I will use Spring to create it.

To *pom.xml's* dependencies I add DBCP and Derby client:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany</groupId>
    <artifactId>MySpringHelloWorld</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.7</maven.compiler.source>
        <maven.compiler.target>1.7</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.0.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-dbcp2</artifactId>
            <version>2.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.derby</groupId>
            <artifactId>derbyclient</artifactId>
            <version>10.11.1.1</version>
        </dependency>
    </dependencies>
</project>
```

I make sure JavaDB (Derby) database server is running. Now in Main I write:

```
package pl.kuku.myspringhelloworld;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;
import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
public class Main {
    public static void main(String[] args) throws Exception {
        BasicDataSource source = new BasicDataSource();
        source.setUrl("jdbc:derby://localhost:1527/sample");
        source.setUsername("app");
        source.setPassword("app");
        Connection connection = source.getConnection();
        Statement st = connection.createStatement();
        ResultSet r = st.executeQuery("select * from customer");
        r.next();
        System.out.println("name = " + r.getString("name"));
        r.next();
        System.out.println("name = " + r.getString("name"));
    }
}
```

I run it, it works.

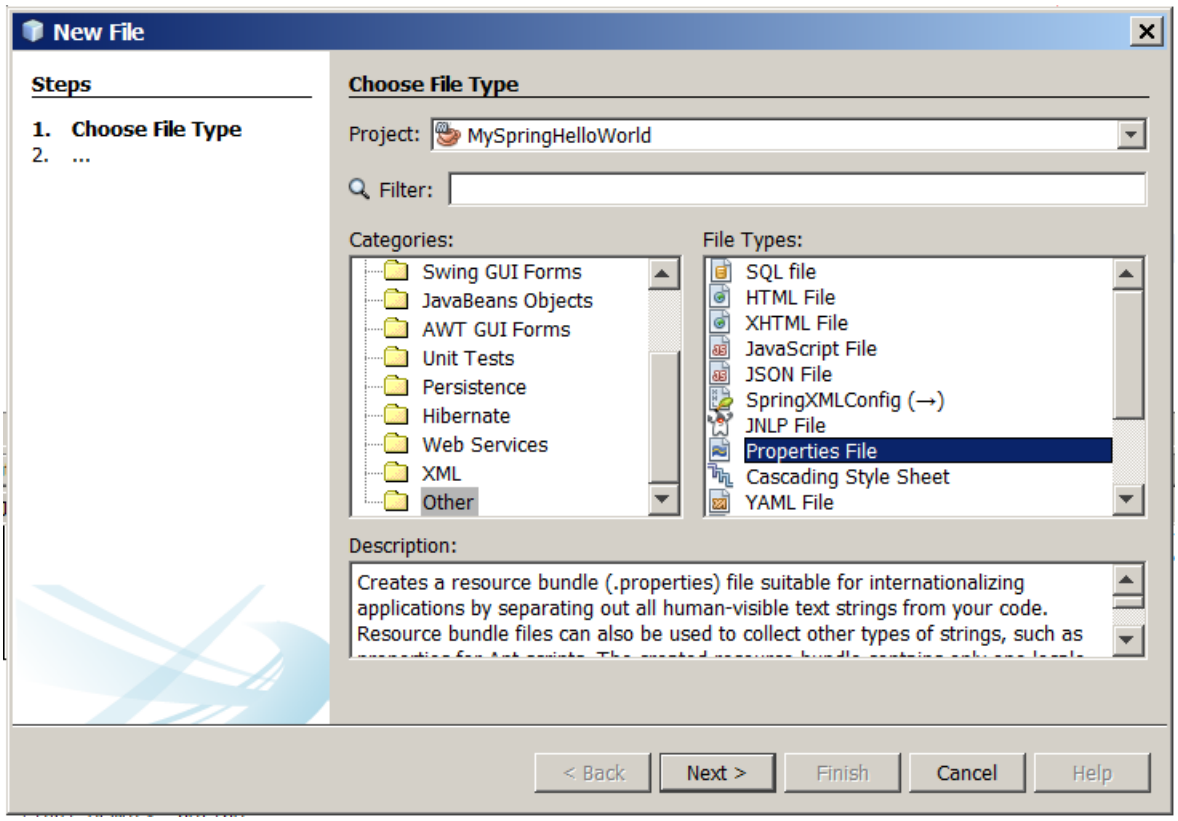
Now I move creating this datasource to *myconfig.xml*. In *myconfig.xml* I write:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"


xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
    <bean id="current_date" class="java.util.Date" />
    <bean id="myproducer"
class="pl.kuku.myspringhelloworld.DateProducer">
        <!-- <property name="date" ref="primaaprilis" /> -->
        <property name="date" ref="current_date" />
    </bean>
    <bean id="myprinter"
class="pl.kuku.myspringhelloworld.DatePrinter" />
    <bean id="mymyprogram"
class="pl.kuku.myspringhelloworld.MyProgram"
        init-method="run"
        p:producer-ref="myproducer"
        p:printer-ref="myprinter"
    />
    <bean id="mydatasource"
class="org.apache.commons.dbcp2.BasicDataSource"
        p:url="jdbc:derby://localhost:1527/sample"
        p:username="app"
        p:password="app"
    />
</beans>
```

I change Main like that:

```
package pl.kuku.myspringhelloworld;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;
import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
public class Main {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("/myconfig.xml");
        BasicDataSource source =
context.getBean(BasicDataSource.class);
        Connection connection = source.getConnection();
        Statement st = connection.createStatement();
        ResultSet r = st.executeQuery("select * from customer");
        r.next();
        System.out.println("name = " + r.getString("name"));
        r.next();
        System.out.println("name = " + r.getString("name"));
    }
}
```

Now I will move connection parameters to parameters file. In *src/main/resources* (in Netbeans it is in *other sources*) I create a new file of type *other -> properties file* and I call it *myproperties.properties*:



In this file I write:

```
jdbc.url = jdbc\:derby\://localhost:1527/sample
jdbc.username = app
jdbc.password = app
```

As we can see, I had to escape colons.

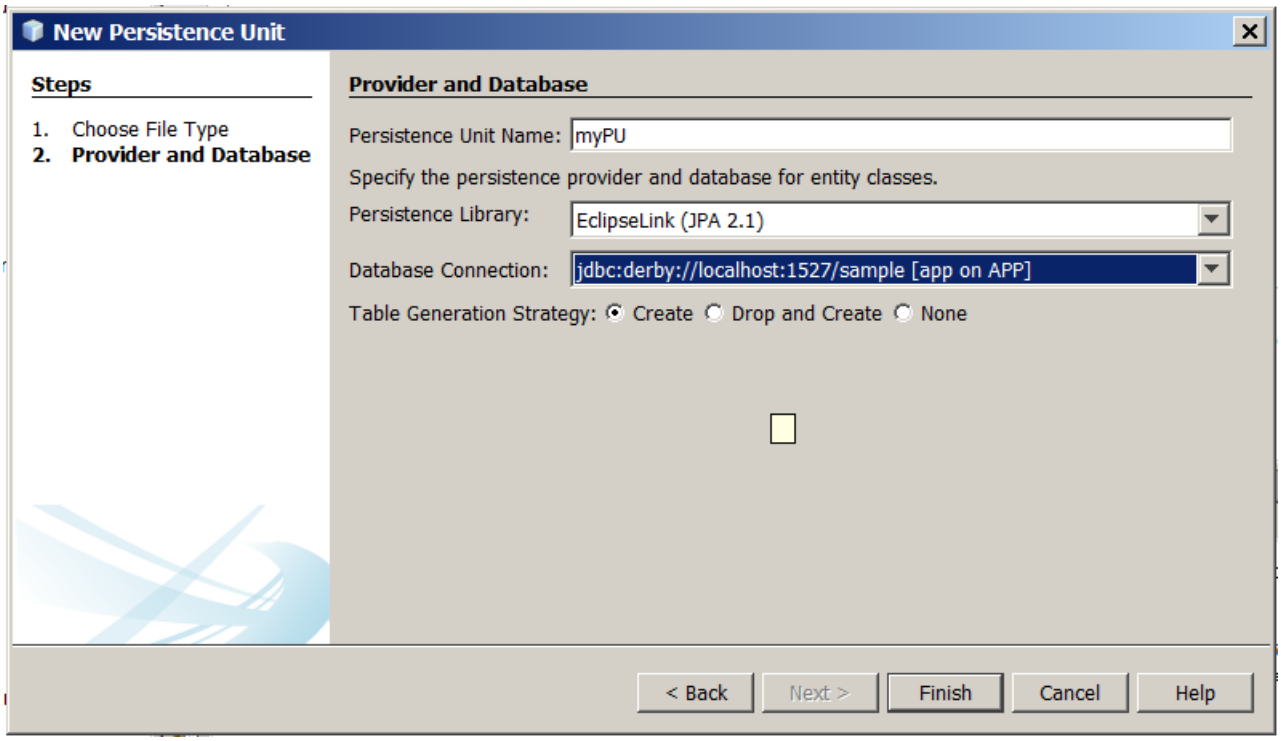
Now in *myconfig.xml* I write:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"


xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
    <bean id="current_date" class="java.util.Date" />
    <bean id="myproducer"
class="pl.kuku.myspringhelloworld.DateProducer">
        <!-- <property name="date" ref="primaaprilis" /> -->
        <property name="date" ref="current_date" />
    </bean>
    <bean id="myprinter"
class="pl.kuku.myspringhelloworld.DatePrinter" />
    <bean id="mymyprogram"
class="pl.kuku.myspringhelloworld.MyProgram"
        init-method="run"
        p:producer-ref="myproducer"
        p:printer-ref="myprinter"
    />
    <context:property-placeholder
location="classpath:myproperties.properties" />
    <bean id="mydatasource"
class="org.apache.commons.dbcp2.BasicDataSource"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}"
    />
</beans>
```

I run the program, it works.

What we have done so far with this datasource can be useful by itself – I could define datasource in Spring config and then use it in DAOs (let's remember that objects created by Spring are singletons). Or even more: I could also create DAOs using Spring and inject them into Swing widgets. But let's be careful – probably it wouldn't work in EDT (depends on how careful would I do that).

Now I create in my project a persistence unit (it is, a *persistence.xml* file):

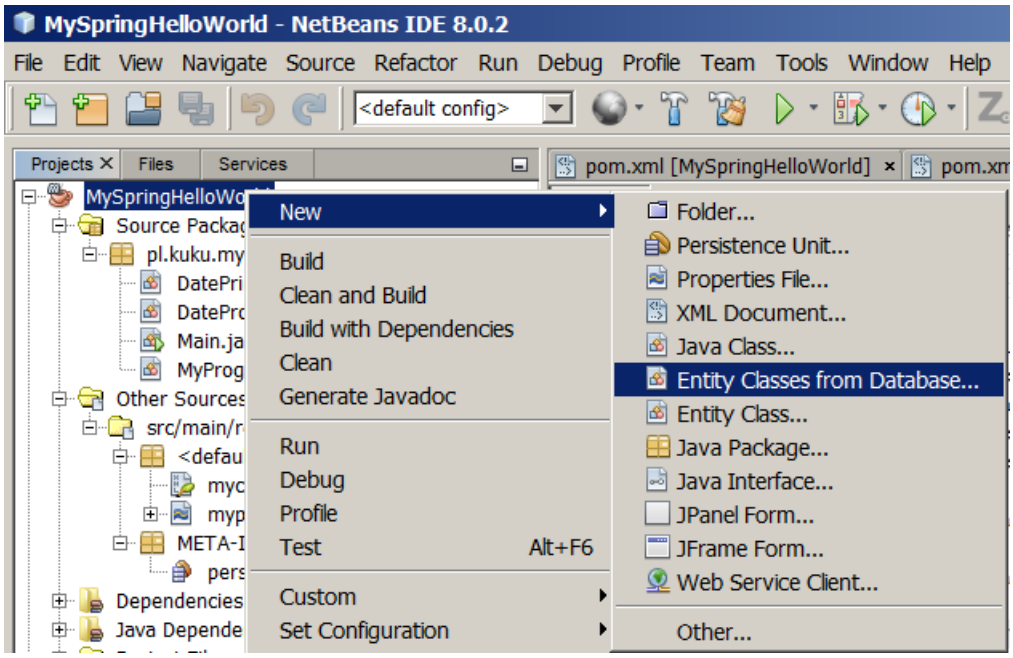


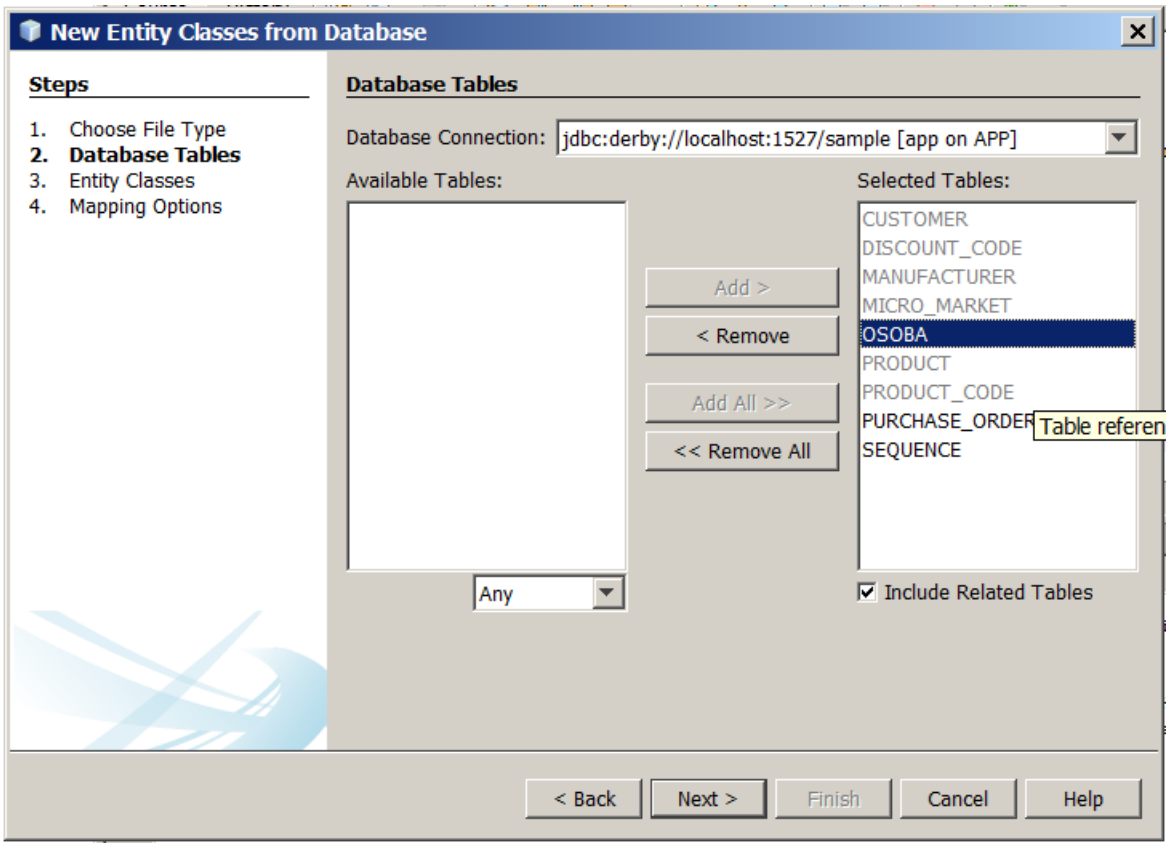
The contents of this *persistence.xml* file is:

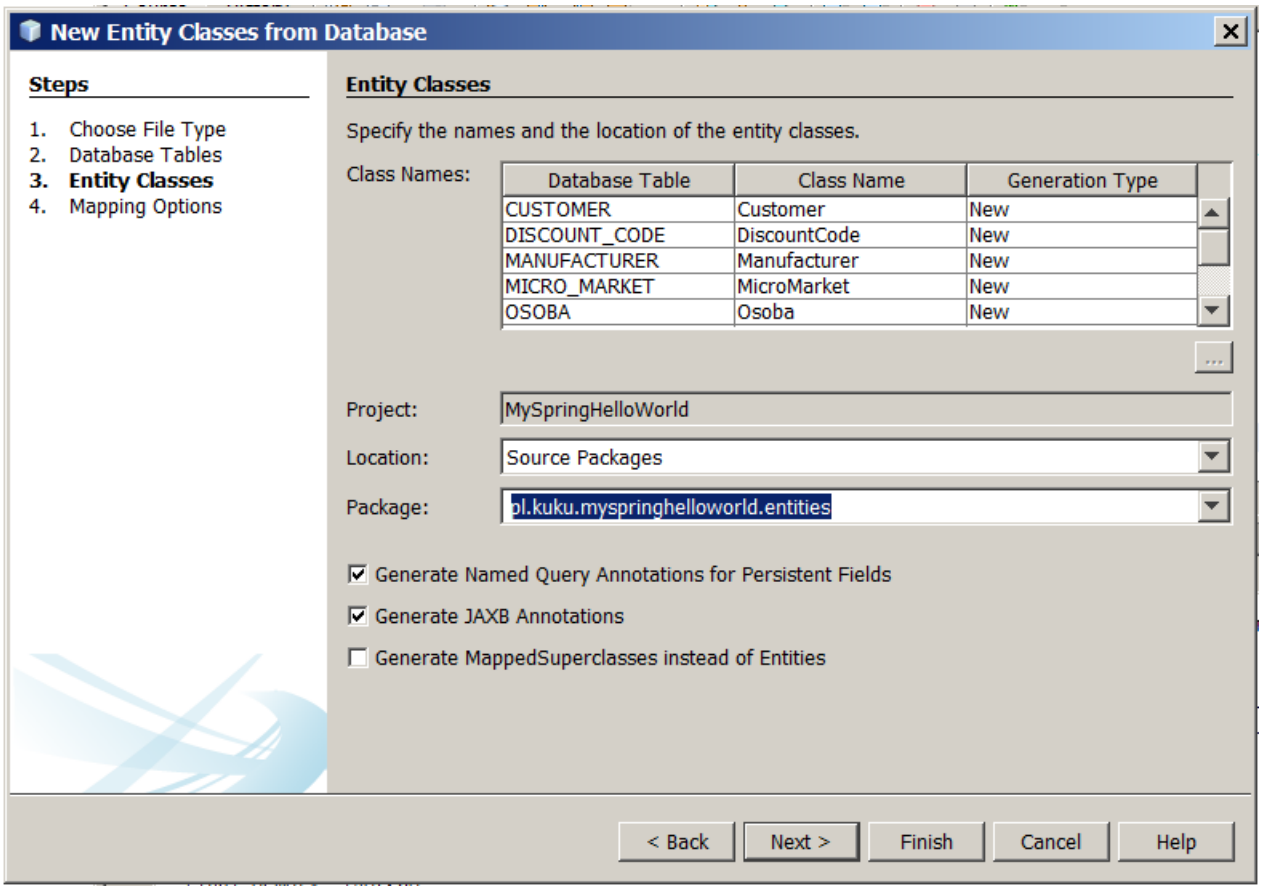
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="myPU" transaction-type="RESOURCE_LOCAL">

<provider>org.eclipse.persistence.jpa.PersistenceProvider</provide
r>
    <properties>
      <property name="javax.persistence.jdbc.url"
value="jdbc:derby://localhost:1527/sample"/>
      <property name="javax.persistence.jdbc.user" value="app"/>
      <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.password"
value="app"/>
      <property name="javax.persistence.schema-
generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

As we can see, I use the connection which is already defined in Eclipse. This is the connection to the sample, testing Derby database which, for instance, contains *Customer* table. I will use that table soon.

Now in my project, in *pl.kuku.myspringhelloworld.entities* package, I create entity classes from the database:




New Entity Classes from Database
Steps
1. Choose File Type
2. Database Tables
3. Entity Classes
4. Mapping Options
Database Tables
Database Connection: jdbc:derby://localhost:1527/sample [app on APP]
Available Tables:
Selected Tables:
CUSTOMER
DISCOUNT_CODE
MANUFACTURER
MICRO_MARKET
OSOBA
PRODUCT
PRODUCT_CODE
PURCHASE_ORDER
SEQUENCE
Table referen
Add >
< Remove
Add All >>
<< Remove All
Any
Include Related Tables
< Back
Next >
Finish
Cancel
Help


New Entity Classes from Database
Steps
1. Choose File Type
2. Database Tables
3. Entity Classes
4. Mapping Options
Entity Classes
Specify the names and the location of the entity classes.
Class Names:
Database Table
Class Name
Generation Type
CUSTOMER
Customer
New
DISCOUNT_CODE
DiscountCode
New
MANUFACTURER
Manufacturer
New
MICRO_MARKET
MicroMarket
New
OSOBA
Osoba
New
Project:
MySpringHelloWorld
Location:
Source Packages
Package:
pl.kuku.myspringhelloworld.entities
Generate Named Query Annotations for Persistent Fields
Generate JAXB Annotations
Generate MappedSuperclasses instead of Entities
< Back
Next >
Finish
Cancel
Help

In Main I write a piece of code which displays names of all customers using EntityManager:

```
package pl.kuku.myspringhelloworld;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
import org.springframework.orm.jpa.LocalEntityManagerFactoryBean;
import pl.kuku.myspringhelloworld.entities.Customer;
public class Main {
    public static void main(String[] args) throws Exception {
        EntityManager em =
Persistence.createEntityManagerFactory("myPU").createEntityManager
();
        List<Customer> result = em.createQuery("select c from
Customer c").getResultList();
        for (Customer c : result) {
            System.out.println(c.getName());
        }
    }
}
```

So, now my program asks the Persistence class to create entity manager factory. This Persistence class takes a look at *persistence.xml* and learns from it that it has to use EclipseLink's persistence provider:

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider</
provider>
```

So it call some method on the *org.eclipse.persistence.jpa.PersistenceProvider*. Then *org.eclipse.persistence.jpa.PersistenceProvider* takes a look at *persistence.xml* and learns from it what JDBC driver it should use to connect to a database etc:

```
<properties>
  <property name="javax.persistence.jdbc.url" value="jdbc:derby://
localhost:1527/sample"/>
  <property name="javax.persistence.jdbc.user" value="app"/>
  <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.ClientDriver"/>
  <property name="javax.persistence.jdbc.password" value="app"/>
  <property name="javax.persistence.schema-
generation.database.action" value="create"/>
</properties>
```

So EclipseLink library uses PostgreSQL JDBC driver to connect to database.

To make sure is is really this way I'm going to debug the program. I write somewhere the line:

```
org.apache.derby.jdbc.ClientDriver tmp;
```

I need it only to open the *org.apache.derby.jdbc.ClientDriver* class in Netbeans. So I ctrl+click this class name so it opens in Netbeans' editor and then I remove this line. In *ClientDriver* I find the *connect* method and I put breakpoint on it. Then I run my program in debug mode and when it stops on the breakpoint I read the stacktrace:

```
org.apache.derby.jdbc.ClientDriver.connect
java.sql.DriverManager.getConnection(DriverManager.java:664)
java.sql.DriverManager.getConnection(DriverManager.java:208)
org.eclipse.persistence.sessions.DefaultConnector.connect(DefaultC
onnector.java:98)
org.eclipse.persistence.sessions.DatasourceLogin.connectToDatasour
ce(DatasourceLogin.java:162)
org.eclipse.persistence.internal.sessions.DatabaseSessionImpl.setO
rDetectDatasource(DatabaseSessionImpl.java:204)
org.eclipse.persistence.internal.sessions.DatabaseSessionImpl.logi
nAndDetectDatasource(DatabaseSessionImpl.java:741)
org.eclipse.persistence.internal.jpa.EntityManagerFactoryProvider.
login(EntityManagerFactoryProvider.java:239)
org.eclipse.persistence.internal.jpa.EntityManagerSetupImpl.deploy
(EntityManagerSetupImpl.java:685)
org.eclipse.persistence.internal.jpa.EntityManagerFactoryDelegate.
getAbstractSession(EntityManagerFactoryDelegate.java:204)
org.eclipse.persistence.internal.jpa.EntityManagerFactoryDelegate.
getDatabaseSession(EntityManagerFactoryDelegate.java:182)
org.eclipse.persistence.internal.jpa.EntityManagerFactoryImpl.getD
atabaseSession(EntityManagerFactoryImpl.java:527)
org.eclipse.persistence.jpa.PersistenceProvider.createEntityManage
rFactoryImpl(PersistenceProvider.java:140)
org.eclipse.persistence.jpa.PersistenceProvider.createEntityManage
rFactory(PersistenceProvider.java:177)
javax.persistence.Persistence.createEntityManagerFactory(Persisten
ce.java:79)
javax.persistence.Persistence.createEntityManagerFactory(Persisten
ce.java:54)
pl.kuku.myspringhelloworld.Main.main(Main.java:57)
```

Main's line 57 mentioned in this stacktrace is:

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("myPU");
```

5.2. how does <property-placeholder /> work

Now let's understand how it is that *property-placeholder* element works. There is nothing special in this tag, Spring by itself does not know this tag. Spring has a general and flexible mechanism that allows anyone to invent his own XML namespace, create its XSD and declare Java classess whose methods will be called when a bean factory will find this element in an XML file. Authors of *property-placeholder* used this mechanism.

In Netbeans I look at *dependencies -> spring-context-VERSION.jar -> spring.handlers*. In this file it is declared what class is responsible for *context* namespace:

```
http\://www.springframework.org/schema/
context=org.springframework.context.config.ContextNamespaceHandler
(...)
```

The source of *org.springframework.context.config.ContextNamespaceHandler* I can find at *grepcode.com* (*http://grepcode.com/file/repository.springsource.com/org.springframework/org.springframework.context/2.5.4/org/springframework/context/config/ContextNamespaceHandler.java*).

This class says:

```
package org.springframework.context.config;
public ContextNamespaceHandler extends NamespaceHandlerSupport {
     public void init() {
          registerBeanDefinitionParser("property-placeholder", new
PropertyPlaceholderBeanDefinitionParser());
          (...)
```

It means that it is the class *PropertyPlaceholderBeanDefinitionParser* that handles *context:property-placeholder* element. This class (I view its sources at grepcode) extends *AbstractPropertyLoadingBeanDefinitionParser*, which looks like that:

```
package org.springframework.context.config;
import org.w3c.dom.Element;
import org.springframework.beans.factory.config.BeanDefinition;
import
org.springframework.beans.factory.support.BeanDefinitionBuilder;
import
org.springframework.beans.factory.xml.AbstractSingleBeanDefinition
Parser;
import org.springframework.util.StringUtils;
abstract class AbstractPropertyLoadingBeanDefinitionParser extends
AbstractSingleBeanDefinitionParser {
  (...)
    @Override
    protected void doParse(Element element, BeanDefinitionBuilder
builder) {
         String location = element.getAttribute("location");
         if (StringUtils.hasLength(location)) {
              String[] locations =
StringUtils.commaDelimitedListToStringArray(location);
              builder.addPropertyValue("locations", locations);
         }
         (...)
```

Here we can see the code that handles *location* attribute of *<context:property-placeholder />* element.

This is how it works.

5.3. InitializingBean interface, interface injection

Before we continue, let's look at one detail.

Sometimes we have some bean which wants to do after it has all properties set. If a bean implements the *InitializingBean* interface and has a method called *afterPropertiesSet*, Spring calls this method after it sets all properties on this bean.

Let's try it.

I create such bean:

```
package pl.kuku.myspringhelloworld;
import org.springframework.beans.factory.InitializingBean;
public class MyBean implements InitializingBean {
    private int x;
    public int getX() {
        return x;
    }
    public void setX(int x) {
        System.out.println("setter is called");
        this.x = x;
    }
    @Override
    public void afterPropertiesSet() {
        System.out.println("method afterPropertiesSet is called");
    }
}
```

In *myconfig.xml* I write:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"


xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="mymybean" class="pl.kuku.myspringhelloworld.MyBean"
        p:x="3"
    />
</beans>
```

In Main I just create the Spring context:

```
package pl.kuku.myspringhelloworld;
import java.sql.Connection;
```

```
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
import org.springframework.orm.jpa.LocalEntityManagerFactoryBean;
import pl.kuku.myspringhelloworld.entities.Customer;
public class Main {
    public static void main(String[] args) throws Exception {
        new ClassPathXmlApplicationContext("/myconfig.xml");
    }
}
```

I run this program. It displays:

```
setter is called
method afterPropertiesSet is called
```

In Spring there are more such interfaces. There are some interfaces – their names use to end with *aware* – such that if a bean implements this interface, Spring calls a method of this interface in some special situation. This mechanizm is called *interface injection*.

5.4. LocalEntityManagerFactoryBean

Now I will create this EntityManagerFactory using Spring. But there is one problem with it: EntityManagerFactory is not a Java bean. There is the reason Spring authors created *LocalEntityManagerFactoryBean*[2] class – it is a bean wrapping EntityManagerFactory.

Well, actually it is also possible to use Spring to create objects that are not beans, but the common practice in Spring is to use this class. FIXME: why?

First I instantiate this class without using Spring's context. This class implements InitializingBean, co after we set its properties we must call *afterPropertiesSet* on it.

2 Once I was interesting why this entity manager factory is called local. I asked about it on Stack Overflow (*http://stackoverflow.com/questions/29283419/why-localentitymanagerfactorybean-class-is-called-local/29289555#29289555*) and somebody has answered:
I dont have an authoritative source, however I'm pretty sure that it is "Local" in that it is local to the spring application context used by the application.
Spring doesn't provide a Remote EMF, however other components such as application servers will. For instance JBoss AS (an open source J2EE app server) can manage JPA EMF's and will make it available to your application at runtime, eg over JNDI, refer to the JBoss docs (https://docs.jboss.org/author/display/AS71/JPA+Reference+Guide#JPAReferenceGuide-Entitymanager)

In Main I write:

```
package pl.kuku.myspringhelloworld;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
import org.springframework.orm.jpa.LocalEntityManagerFactoryBean;
import pl.kuku.myspringhelloworld.entities.Customer;
public class Main {
    public static void main(String[] args) throws Exception {
        LocalEntityManagerFactoryBean lemfb = new
LocalEntityManagerFactoryBean();
        lemfb.setPersistenceUnitName("myPU");
        lemfb.afterPropertiesSet();
        EntityManagerFactory emf =
lemfb.getNativeEntityManagerFactory();
        EntityManager em = emf.createEntityManager();
        List<Customer> result = em.createQuery("select c from
Customer c").getResultList();
        for (Customer c : result) {
            System.out.println(c.getName());
        }
    }
}
```

I run this program. It works.

Now I move instantiation and configuration of LocalEntityManagerFactoryBean to *myconfig.xml*. In *myconfig.xml* I write:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"


xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
```

```
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
    <bean id="current_date" class="java.util.Date" />
    <bean id="myproducer"
class="pl.kuku.myspringhelloworld.DateProducer">
        <!-- <property name="date" ref="primaaprilis" /> -->
        <property name="date" ref="current_date" />
    </bean>
    <bean id="myprinter"
class="pl.kuku.myspringhelloworld.DatePrinter" />
    <bean id="mymyprogram"
class="pl.kuku.myspringhelloworld.MyProgram"
        init-method="run"
        p:producer-ref="myproducer"
        p:printer-ref="myprinter"
    />
    <!--
    <bean id="mydatasource"
class="org.apache.commons.dbcp2.BasicDataSource"
        p:url="jdbc:derby://localhost:1527/sample"
        p:username="app"
        p:password="app"
    />
    -->
    <context:property-placeholder
location="classpath:myproperties.properties" />
    <bean id="mydatasource"
class="org.apache.commons.dbcp2.BasicDataSource"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}"
    />
<!--    <bean id="mymybean"
class="pl.kuku.myspringhelloworld.MyBean"
        p:x="3"
    /> -->
    <bean
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean"
        p:persistenceUnitName="myPU"
    />
</beans>
```

In Main I write:

```
package pl.kuku.myspringhelloworld;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
import org.springframework.orm.jpa.LocalEntityManagerFactoryBean;
import pl.kuku.myspringhelloworld.entities.Customer;
public class Main {
    public static void main(String[] args) throws Exception {
        ApplicationContext context = new
ClassPathXmlApplicationContext("/myconfig.xml");
        LocalEntityManagerFactoryBean lemfb =
context.getBean(LocalEntityManagerFactoryBean.class);
        EntityManagerFactory emf =
lemfb.getNativeEntityManagerFactory();
        EntityManager em = emf.createEntityManager();
        List<Customer> result = em.createQuery("select c from
Customer c").getResultList();
        for (Customer c : result) {
            System.out.println(c.getName());
        }
    }
}
```

I run this program. It works.

5.5. now I move as much code as possible out of Main

Now I will refactor my program to be more modular. The code that gets data from database and displays it I will move away from Main. So Main will be just a oneliner creating Spring context.

I create a class called *DBDataDisplayer*:

```
package pl.kuku.myspringhelloworld;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import org.springframework.orm.jpa.LocalEntityManagerFactoryBean;
import pl.kuku.myspringhelloworld.entities.Customer;
public class DBDataDisplayer {
    private LocalEntityManagerFactoryBean lemfb;
    public LocalEntityManagerFactoryBean getLemfb() {
        return lemfb;
    }
    public void setLemfb(LocalEntityManagerFactoryBean lemfb) {
        this.lemfb = lemfb;
    }
    public void doWhatMustBeDone() {
        EntityManagerFactory emf =
lemfb.getNativeEntityManagerFactory();
        EntityManager em = emf.createEntityManager();
        List<Customer> result = em.createQuery("select c from
Customer c").getResultList();
        for (Customer c : result) {
            System.out.println(c.getName());
        }
    }
}
```

In Main I write:

```
package pl.kuku.myspringhelloworld;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
import org.springframework.orm.jpa.LocalEntityManagerFactoryBean;
import pl.kuku.myspringhelloworld.entities.Customer;
public class Main {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext cx = new
ClassPathXmlApplicationContext("/myconfig.xml");
    }
}
```

In *myconfig.xml* I wrote:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"


xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
         http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
    <bean id="current_date" class="java.util.Date" />
    <bean id="myproducer"
class="pl.kuku.myspringhelloworld.DateProducer">
        <!-- <property name="date" ref="primaaprilis" /> -->
        <property name="date" ref="current_date" />
```

```
    </bean>
    <bean id="myprinter"
class="pl.kuku.myspringhelloworld.DatePrinter" />
    <bean id="mymyprogram"
class="pl.kuku.myspringhelloworld.MyProgram"
        init-method="run"
        p:producer-ref="myproducer"
        p:printer-ref="myprinter"
    />
    <!--
    <bean id="mydatasource"
class="org.apache.commons.dbcp2.BasicDataSource"
        p:url="jdbc:derby://localhost:1527/sample"
        p:username="app"
        p:password="app"
    />
    -->
    <context:property-placeholder
location="classpath:myproperties.properties" />
    <bean id="mydatasource"
class="org.apache.commons.dbcp2.BasicDataSource"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}"
    />
<!--    <bean id="mymybean"
class="pl.kuku.myspringhelloworld.MyBean"
        p:x="3"
    /> -->
    <bean id="mylemfb"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean"
        p:persistenceUnitName="myPU"
    />
    <bean class="pl.kuku.myspringhelloworld.DBDataDisplayer"
        p:lemfb-ref="mylemfb"
        init-method="doWhatMustBeDone"
    />
</beans>
```

However, when I try to run this program, it throws an exception:

```
(...)
Caused by: java.lang.IllegalStateException: Cannot convert value
of type [com.sun.proxy.$Proxy3 implementing
javax.persistence.EntityManagerFactory,javax.persistence.Persisten
ceUnitUtil,org.eclipse.persistence.jpa.JpaEntityManagerFactory,org
.springframework.orm.jpa.EntityManagerFactoryInfo] to required
type [org.springframework.orm.jpa.LocalEntityManagerFactoryBean]
for property 'lemfb': no matching editors or conversion strategy
found
(...)
```

It seems that the object that Spring tries to put into the lemfb attribute of DBDataDisplayer is not an object of type *LocalEntityManagerFactoryBean*. Let's investigate it.

First I comment out this fragment of *myconfig.xml*, so my program runs:

```
(...)
    <bean id="mylemfb"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean"
        p:persistenceUnitName="myPU"
    />
    <!--
    <bean class="pl.kuku.myspringhelloworld.DBDataDisplayer"
        p:lemfb-ref="mylemfb"
        init-method="doWhatMustBeDone"
    />
    -->
</beans>
```

In Main I write such test:

```
package pl.kuku.myspringhelloworld;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
import org.springframework.orm.jpa.LocalEntityManagerFactoryBean;
import pl.kuku.myspringhelloworld.entities.Customer;
public class Main {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("/myconfig.xml");
        Object b1 = ctx.getBean("mylemfb");
        System.out.println("first bean type is:" +
b1.getClass().getName() + ", this type instanceof
LocalEntityManagerFactoryBean: " + (b1 instanceof
LocalEntityManagerFactoryBean) + ", this type instanceof
EntityManagerFactory: " + (b1 instanceof EntityManagerFactory));
        Object b2 =
ctx.getBean(LocalEntityManagerFactoryBean.class);
        System.out.println("second bean type is " +
b2.getClass().getName() + ", this type instanceof
LocalEntityManagerFactoryBean: " + (b2 instanceof
LocalEntityManagerFactoryBean) + ", this type instanceof
EntityManagerFactory: " + (b2 instanceof EntityManagerFactory));
    }
}
```

I run this program and it prints:

```
first bean type is:com.sun.proxy.$Proxy3, this type instanceof
LocalEntityManagerFactoryBean: false, this type instanceof
EntityManagerFactory: true
second bean type is
org.springframework.orm.jpa.LocalEntityManagerFactoryBean, this
type instanceof LocalEntityManagerFactoryBean: true, this type
instanceof EntityManagerFactory: false
```

It means that – due to some magic, done with interface injection – when I try to get the bean using class, I get the object of type LocalEntityManagerFactoryBean (so everything is normal), but when I try to get the bean using name, I get another object, which is of type com.sun.proxy.$Proxy3. This class com.sun.proxy.$Proxy3 does not extend LocalEntityManagerFactoryBean but it implements EntityManagerFactory.

So Spring tries to inject an EntityManagerFactory instance into DBDataDisplayer. Well, that's even better.

I change DBDataDisplayer:

```
package pl.kuku.myspringhelloworld;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import org.springframework.orm.jpa.LocalEntityManagerFactoryBean;
import pl.kuku.myspringhelloworld.entities.Customer;
public class DBDataDisplayer {
    private EntityManagerFactory emf;
    public EntityManagerFactory getEmf() {
        return emf;
    }
    public void setEmf(EntityManagerFactory emf) {
        this.emf = emf;
    }
    public void doWhatMustBeDone() {
//        EntityManagerFactory emf =
lemfb.getNativeEntityManagerFactory();
        EntityManager em = emf.createEntityManager();
        List<Customer> result = em.createQuery("select c from
Customer c").getResultList();
        for (Customer c : result) {
            System.out.println(c.getName());
        }
    }
}
```

Then in *myconfig.xml* I uncomment the definition of pl.kuku.myspringhelloworld.DBDataDisplayer and modify it a little:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"


xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
```

```
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
    <bean id="current_date" class="java.util.Date" />
    <bean id="myproducer"
class="pl.kuku.myspringhelloworld.DateProducer">
        <!-- <property name="date" ref="primaaprilis" /> -->
        <property name="date" ref="current_date" />
    </bean>
    <bean id="myprinter"
class="pl.kuku.myspringhelloworld.DatePrinter" />
    <bean id="mymyprogram"
class="pl.kuku.myspringhelloworld.MyProgram"
        init-method="run"
        p:producer-ref="myproducer"
        p:printer-ref="myprinter"
    />
    <!--
    <bean id="mydatasource"
class="org.apache.commons.dbcp2.BasicDataSource"
        p:url="jdbc:derby://localhost:1527/sample"
        p:username="app"
        p:password="app"
    />
    -->
    <context:property-placeholder
location="classpath:myproperties.properties" />
    <bean id="mydatasource"
class="org.apache.commons.dbcp2.BasicDataSource"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}"
    />
<!--    <bean id="mymybean"
class="pl.kuku.myspringhelloworld.MyBean"
        p:x="3"
    /> -->
    <bean id="mylemfb"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean"
        p:persistenceUnitName="myPU"
    />
    <bean class="pl.kuku.myspringhelloworld.DBDataDisplayer"
        p:emf-ref="mylemfb"
        init-method="doWhatMustBeDone"
    />
</beans>
```

In Main I write:

```
package pl.kuku.myspringhelloworld;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext
;
import org.springframework.orm.jpa.LocalEntityManagerFactoryBean;
import pl.kuku.myspringhelloworld.entities.Customer;
public class Main {
    public static void main(String[] args) throws Exception {
        new ClassPathXmlApplicationContext("/myconfig.xml");
    }
}
```

I run the program. It works.

5.6. postprocessors

Soon I will further simplify my program. But before we have to look at one mechanism.

I create class *MyBeanPostProcessor*:

```
package pl.kuku.myspringhelloworld;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
public class MyBeanPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization(Object o, String
name) throws BeansException {
        System.out.println("postProcessBeforeInitialization is
called with name=" + name + " and object=" + o);
        return o;
    }
    @Override
    public Object postProcessAfterInitialization(Object o, String
name) throws BeansException {
        System.out.println("postProcessAfterInitialization is
called with name=" + name + " and object=" + o);
        return o;
    }
}
```

In *myconfig.xml* I register this class:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
     (...)
    <bean class="pl.kuku.myspringhelloworld.MyBeanPostProcessor" /
>
</beans>
```

I run the program. This is the result:

```
postProcessBeforeInitialization is called with name=primaaprilis
and object=Wed Apr 01 13:43:23 CEST 2015
postProcessAfterInitialization is called with name=primaaprilis
and object=Wed Apr 01 13:43:23 CEST 2015
postProcessBeforeInitialization is called with name=current_date
and object=Sun Mar 29 13:43:23 CEST 2015
postProcessAfterInitialization is called with name=current_date
and object=Sun Mar 29 13:43:23 CEST 2015
postProcessBeforeInitialization is called with name=myproducer and
object=pl.kuku.myspringhelloworld.DateProducer@24f60b
postProcessAfterInitialization is called with name=myproducer and
object=pl.kuku.myspringhelloworld.DateProducer@24f60b
postProcessBeforeInitialization is called with name=myprinter and
object=pl.kuku.myspringhelloworld.DatePrinter@1e2203c
postProcessAfterInitialization is called with name=myprinter and
object=pl.kuku.myspringhelloworld.DatePrinter@1e2203c
postProcessBeforeInitialization is called with name=mymyprogram
and object=pl.kuku.myspringhelloworld.MyProgram@b6cbcc
date: Sun Mar 29 13:43:23 CEST 2015
postProcessAfterInitialization is called with name=mymyprogram and
```

```
object=pl.kuku.myspringhelloworld.MyProgram@b6cbcc
postProcessBeforeInitialization is called with name=mydatasource
and object=org.apache.commons.dbcp2.BasicDataSource@33cc44
postProcessAfterInitialization is called with name=mydatasource
and object=org.apache.commons.dbcp2.BasicDataSource@33cc44
mar 29, 2015 1:43:23 PM
org.springframework.orm.jpa.LocalEntityManagerFactoryBean
createNativeEntityManagerFactory
INFO: Building JPA EntityManagerFactory for persistence unit
'myPU'
postProcessBeforeInitialization is called with name=mylemfb and
object=org.springframework.orm.jpa.LocalEntityManagerFactoryBean@1
c7e1bb
[EL Info]: 2015-03-29 13:43:26.205--ServerSession(2297529)--
EclipseLink, version: Eclipse Persistence Services -
2.5.2.v20140319-9ad6abd
[EL Info]: connection: 2015-03-29 13:43:28.921--
ServerSession(2297529)--file:/C:/Users/admin/Documents/
NetBeansProjects/MySpringHelloWorld/target/classes/_myPU login
successful
postProcessAfterInitialization is called with name=mylemfb and
object=org.springframework.orm.jpa.LocalEntityManagerFactoryBean@1
c7e1bb
postProcessAfterInitialization is called with name=mylemfb and
object=org.springframework.orm.jpa.LocalEntityManagerFactoryBean@1
c7e1bb
postProcessBeforeInitialization is called with
name=pl.kuku.myspringhelloworld.DBDataDisplayer#0 and
object=pl.kuku.myspringhelloworld.DBDataDisplayer@13b58ca
Jumbo Eagle Corp
New Enterprises
Wren Computers
Small Bill Company
Bob Hosting Corp.
Early CentralComp
John Valley Computers
Big Network Systems
West Valley Inc.
Zed Motor Co
Big Car Parts
Old Media Productions
Yankee Computer Repair Ltd
postProcessAfterInitialization is called with
name=pl.kuku.myspringhelloworld.DBDataDisplayer#0 and
object=pl.kuku.myspringhelloworld.DBDataDisplayer@13b58ca
```

As we can see, for each bean both methods of MyBeanPostProcessor are called. One is called before the bean's object is configured and one after. This is how *BeanPostProcessor* interface work. It is not the only Spring's postprocessor – another is,

for instance, BeanFactoryPostProcessor (*http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/config/BeanFactoryPostProcessor.html*).

5.7. if we didn't use ClassPathXmlApplicationContext we would have to do more to make this example work

The class *ClassPathXmlApplicationContext* is not the only Spring bean container. There are more basic containers, like XmlBeanFactory. All Spring bean containers implement BeanFactory interface. In normal circumstances there is no reason not to use ClassPathXmlApplicationContext, but it is good to understand what ClassPathXmlApplicationContext gives us.

So, let's try to use XmlBeanFactory instead of ClassPathXmlApplicationContext. I leave *myconfig.xml* as it is:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"


xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
    <bean id="current_date" class="java.util.Date" />
    <bean id="myproducer"
class="pl.kuku.myspringhelloworld.DateProducer">
        <!-- <property name="date" ref="primaaprilis" /> -->
        <property name="date" ref="current_date" />
    </bean>
    <bean id="myprinter"
class="pl.kuku.myspringhelloworld.DatePrinter" />
    <bean id="mymyprogram"
class="pl.kuku.myspringhelloworld.MyProgram"
        init-method="run"
        p:producer-ref="myproducer"
        p:printer-ref="myprinter"
    />
    <!--
    <bean id="mydatasource"
class="org.apache.commons.dbcp2.BasicDataSource"
        p:url="jdbc:derby://localhost:1527/sample"
        p:username="app"
        p:password="app"
    />
    -->
    <context:property-placeholder
location="classpath:myproperties.properties" />
    <bean id="mydatasource"
class="org.apache.commons.dbcp2.BasicDataSource"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
```

```
        p:password="${jdbc.password}"
    />
<!--    <bean id="mymybean"
class="pl.kuku.myspringhelloworld.MyBean"
        p:x="3"
    />-->
    <bean id="mylemfb"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean"
        p:persistenceUnitName="myPU"
    />
    <bean class="pl.kuku.myspringhelloworld.DBDataDisplayer"
        p:emf-ref="mylemfb"
        init-method="doWhatMustBeDone"
    />
    <bean class="pl.kuku.myspringhelloworld.MyBeanPostProcessor" /
>
</beans>
```

Details are not very important in this experiment – what is important is just that I define some beans.

In Main I write:

```
package pl.kuku.myspringhelloworld;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
public class Main {
    public static void main(String[] args) throws Exception {
        XmlBeanFactory xbf = new XmlBeanFactory(new
ClassPathResource("myconfig.xml"));
    }
}
```

I run the program. Nothing happens.

It is because ClassPathXmlApplicationContext – which we used before – create all beans' objects on start, but XmlBeanFactory – which we use now – creates them only on demans. So I add one line to Main:

```
package pl.kuku.myspringhelloworld;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
public class Main {
    public static void main(String[] args) throws Exception {
        XmlBeanFactory xbf = new XmlBeanFactory(new
ClassPathResource("myconfig.xml"));
        xbf.getBean(DBDataDisplayer.class);
    }
}
```

I run the program. It displays:

```
Jumbo Eagle Corp
New Enterprises
Wren Computers
Small Bill Company
Bob Hosting Corp.
Early CentralComp
John Valley Computers
Big Network Systems
West Valley Inc.
Zed Motor Co
Big Car Parts
Old Media Productions
Yankee Computer Repair Ltd
```

As we can see, now DBDataDisplayer works. But MyBeanPostProcessor does not work. It is because in ClassPathXmlApplicationContext – which we used before – bean postprocessors work if they are registered in the config file, but in XmlBeanFactory – which we use now – we must explicitly imperatively register them. So I add one more line in Main:

```
package pl.kuku.myspringhelloworld;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
public class Main {
    public static void main(String[] args) throws Exception {
        XmlBeanFactory xbf = new XmlBeanFactory(new
ClassPathResource("myconfig.xml"));
        xbf.addBeanPostProcessor(new MyBeanPostProcessor());
        xbf.getBean(DBDataDisplayer.class);
    }
}
```

I remove (or just comment out) my postprocessor's definition from *myconfig.xml* – it is not necessary there any more:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
     (...)
    <!--<bean
class="pl.kuku.myspringhelloworld.MyBeanPostProcessor" />-->
</beans>
```

Now I run this program. It displays:

```
mar 29, 2015 4:56:20 PM
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource
[myconfig.xml]
postProcessBeforeInitialization is called with name=mylemfb and
object=org.springframework.orm.jpa.LocalEntityManagerFactoryBean@e
e68d8
```

```
mar 29, 2015 4:56:20 PM
org.springframework.orm.jpa.LocalEntityManagerFactoryBean
createNativeEntityManagerFactory
INFO: Building JPA EntityManagerFactory for persistence unit
'myPU'
[EL Info]: 2015-03-29 16:56:24.043--ServerSession(24140775)--
EclipseLink, version: Eclipse Persistence Services -
2.5.2.v20140319-9ad6abd
[EL Info]: connection: 2015-03-29 16:56:25.243--
ServerSession(24140775)--file:/C:/Users/admin/Documents/
NetBeansProjects/MySpringHelloWorld/target/classes/_myPU login
successful
postProcessAfterInitialization is called with name=mylemfb and
object=org.springframework.orm.jpa.LocalEntityManagerFactoryBean@e
e68d8
postProcessAfterInitialization is called with name=mylemfb and
object=org.springframework.orm.jpa.LocalEntityManagerFactoryBean@e
e68d8
postProcessBeforeInitialization is called with
name=pl.kuku.myspringhelloworld.DBDataDisplayer#0 and
object=pl.kuku.myspringhelloworld.DBDataDisplayer@1ed560f
Jumbo Eagle Corp
New Enterprises
Wren Computers
Small Bill Company
Bob Hosting Corp.
Early CentralComp
John Valley Computers
Big Network Systems
West Valley Inc.
Zed Motor Co
Big Car Parts
Old Media Productions
Yankee Computer Repair Ltd
postProcessAfterInitialization is called with
name=pl.kuku.myspringhelloworld.DBDataDisplayer#0 and
object=pl.kuku.myspringhelloworld.DBDataDisplayer@1ed560f
```

It means that my postprocessor works.

5.8. autowiring with annotations

The process of putting some objects' references into other objects' attributes – so objects has each other references – is called in Spring *wiring*. There is a postprocessor which allows to do wiring by annotations. This postprocessor is called *org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor*. I will use it.

I edit DBDataDisplayer adding *Autowired* annotation to the *emf* attribute:

```
package pl.kuku.myspringhelloworld;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import pl.kuku.myspringhelloworld.entities.Customer;
public class DBDataDisplayer {
    @Autowired
    private EntityManagerFactory emf;
    public void doWhatMustBeDone() {
        EntityManager em = emf.createEntityManager();
        List<Customer> result = em.createQuery("select c from
Customer c").getResultList();
        for (Customer c : result) {
            System.out.println(c.getName());
        }
    }
}
```

Then I edit *myconfig.xml*. I remove *p:emf-ref* from the definition of DBDataDisplayer's bean and I add the definition of AutowiredAnnotationBeanPostProcessor's bean:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"


xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <context:property-placeholder
location="classpath:myproperties.properties" />
    <bean id="mydatasource"
class="org.apache.commons.dbcp2.BasicDataSource"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}"
    />
    <bean id="mylemfb"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean"
        p:persistenceUnitName="myPU"
    />
    <bean class="pl.kuku.myspringhelloworld.DBDataDisplayer"
        init-method="doWhatMustBeDone"
    />
    <bean
class="org.springframework.beans.factory.annotation.AutowiredAnnot
ationBeanPostProcessor" />
</beans>
```

I run the program. It works.

5.9. some experiments with injecting using annotations

Now I will do some experiments with injecting using annotations.

As we remember, several chapters ago I had two beans of class Date. I put them again in *myconfig.xml*:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
    <bean id="current_date" class="java.util.Date" />
</beans>
```

Now I create a simple class into which I will inject a date, I call it HereIWillInjectADate:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
import org.springframework.beans.factory.annotation.Autowired;
public class HereIWillInjectADate {
    @Autowired
    private Date date;
    public void callMe() {
        System.out.println("date=" + date);
    }
}
```

As we can see, I use *@Autowired* to inject any bean of type Date into the *date* attribute.

Now in *myconfig.xml* I define a bean of type *HereIWillInjectADate* and add *AutowiredAnnotationBeanPostProcessor*:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
    <bean id="current_date" class="java.util.Date" />
    <bean class="pl.kuku.myspringhelloworld.HereIWillInjectADate"
        init-method="callMe"
    />
    <bean
class="org.springframework.beans.factory.annotation.AutowiredAnnot
ationBeanPostProcessor" />
</beans>
```

I try to run the program, but I get an exception:

```
Caused by:
org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type [java.util.Date] is defined: expected
single matching bean but found 2: primaaprilis,current_date
```

It is because in HereIWillInjectADate I ask to inject a bean of type Date, but I have two beans of this type.

According to stackoverflow and many other tutorials and docs I could provide the bean name using *@Qualifier*. I try to do it in HereIWillInjectADate*:*

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
public class HereIWillInjectADate {
    @Autowired
    @Qualifier("primaaprilis")
    private Date date;
    public void callMe() {
        System.out.println("date=" + date);
    }
}
```

But it doesn't work – I still get the same exception:

```
Caused by:
org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type [java.util.Date] is defined: expected
single matching bean but found 2: primaaprilis,current_date
```

However, if the name of the attribute matches the name of some bean, this bean is injected. I edit HereIWillInjectADate:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
import org.springframework.beans.factory.annotation.Autowired;
public class HereIWillInjectADate {
    @Autowired
    private Date primaaprilis;
    public void callMe() {
        System.out.println("date=" + primaaprilis);
    }
}
```

I run the program and it works, it displays:

```
date=Wed Apr 01 11:44:45 CEST 2015
```

Now I comment out (from *myconfig.xml*) one bean of type Date, so I have only one bean of type Date:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="primaaprilis" class="java.util.Date">
        <property name="year" value="115" />
        <property name="month" value="3" />
        <property name="date" value="1" />
    </bean>
    <bean class="pl.kuku.myspringhelloworld.HereIWillInjectADate"
        init-method="callMe"
    />
    <bean
class="org.springframework.beans.factory.annotation.AutowiredAnnot
ationBeanPostProcessor" />
</beans>
```

In *HereIWillInjectADate* I again rename the attribute:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
import org.springframework.beans.factory.annotation.Autowired;
public class HereIWillInjectADate {
    @Autowired
    private Date date;
    public void callMe() {
        System.out.println("date=" + date);
    }
}
```

I run the program. It works without problems – there is only one bean of type Date and I inject it.

Spring also understands standard java annotation *@Inject*.

I edit *pom.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany</groupId>
    <artifactId>MySpringHelloWorld</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.7</maven.compiler.source>
        <maven.compiler.target>1.7</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>4.0.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.0.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-dbcp2</artifactId>
            <version>2.1</version>
        </dependency>
```

```
        <dependency>
            <groupId>org.apache.derby</groupId>
            <artifactId>derbyclient</artifactId>
            <version>10.11.1.1</version>
        </dependency>
        <dependency>
            <groupId>org.eclipse.persistence</groupId>
            <artifactId>eclipselink</artifactId>
            <version>2.5.2</version>
        </dependency>
        <dependency>
            <groupId>org.eclipse.persistence</groupId>

<artifactId>org.eclipse.persistence.jpa.modelgen.processor</artifa
ctId>
            <version>2.5.2</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.inject</groupId>
            <artifactId>javax.inject</artifactId>
            <version>1</version>
        </dependency>
    </dependencies>
</project>
```

I edit *HereIWillInjectADate*:

```
package pl.kuku.myspringhelloworld;
import java.util.Date;
import javax.inject.Inject;
public class HereIWillInjectADate {
    @Inject
    private Date date;
    public void callMe() {
        System.out.println("date=" + date);
    }
}
```

I run the program. It works.

5.10. turning on AutowiredAnnotationBeanPostProcessor with context:annotation-config

Now I come back to my program using LocalEntityManagerFactoryBean. In *myconfig.xml* I put everything this program needs:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="mylemfb"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean"
        p:persistenceUnitName="myPU"
    />
    <bean class="pl.kuku.myspringhelloworld.DBDataDisplayer"
        init-method="doWhatMustBeDone"
    />
    <bean
class="org.springframework.beans.factory.annotation.AutowiredAnnot
ationBeanPostProcessor" />
</beans>
```

I make sure DBDataDisplayer looks like that:

```
package pl.kuku.myspringhelloworld;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import pl.kuku.myspringhelloworld.entities.Customer;
public class DBDataDisplayer {
    @Autowired
    private EntityManagerFactory emf;
    public void doWhatMustBeDone() {
        EntityManager em = emf.createEntityManager();
        List<Customer> result = em.createQuery("select c from
Customer c").getResultList();
        for (Customer c : result) {
            System.out.println(c.getName());
        }
```

```
    }
}
```

I run the program. It works.

When I want to do autowiring with annotations, I don't have to declare AutowiredAnnotationBeanPostProcessor bean – there is a shortcut for it. It uses the same mechanism as context:property-placeholder. Instead of declaring AutowiredAnnotationBeanPostProcessor bean in Spring's XML configuration file I can put there element *<context:annotation-config />*.

So, in *myconfig.xml* I write:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="mylemfb"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean"
        p:persistenceUnitName="myPU"
    />
    <bean class="pl.kuku.myspringhelloworld.DBDataDisplayer"
        init-method="doWhatMustBeDone"
    />
    <context:annotation-config />
</beans>
```

I run the program. It works.

5.11. context:component-scan

There is also another postprocessor, switched on by XML element *context:component-scan*. This postprocessor does what *context:annotation-config* do – injects bean objects in places annotated with *@Autowired*. It also does something more: it scans packages, finds classes annotated with special annotation and registers them in the application context.

I will use *context:component-scan* in my program. I have to annotate DBDataDisplayer with some annotation. I can annotate it with *@org.springframework.stereotype.Component* or with any annotation metaannotated with *@org.springframework.stereotype.Component*. There are three standard Spring annotations metaannotated with *@org.springframework.stereotype.Component*: *@Repository*, *@Controller* and *@Named*.

The postprocessor treats all these four annotations exactly the same[3], but they have different meaning – I mean, there is a convention which says for what types of classes what annotation should be used (source: *http://stackoverflow.com/questions/6827752/whats-the-difference-between-component-repository-service-annotations-in*):

- @Component - generic stereotype for any Spring-managed component,
- @Repository - stereotype for persistence layer,
- @Service - stereotype for service layer,
- @Controller - stereotype for presentation layer (spring-mvc).

There is also another annotation that Spring understands. If annotate a method with a standard Java *@javax.annotation.PostConstruct* annotation, the method will be run after this bean is configured.

So I edit DBDataDisplayer:

```
package pl.kuku.myspringhelloworld;
import java.util.List;
import javax.annotation.PostConstruct;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import pl.kuku.myspringhelloworld.entities.Customer;
@Component
public class DBDataDisplayer {
    @Autowired
    private EntityManagerFactory emf;
    @PostConstruct
    public void doWhatMustBeDone() {
        EntityManager em = emf.createEntityManager();
        List<Customer> result = em.createQuery("select c from
Customer c").getResultList();
        for (Customer c : result) {
            System.out.println(c.getName());
        }
    }
}
```

3 It is true that the postprocessor enabled by *context:component-scan* treats all those annotations the same – but other code may treat them differently. The only difference between them which I know about is that RequestMappingHandlerMapping maps incoming requests only to those beans which are annotated with *@Controller* (and not, for instance, *@Component*) or *@RequestMapping*.

I also edit *myconfig.xml*:

```
<?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"
       xmlns:p="http://www.springframework.org/schema/p"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean"
        p:persistenceUnitName="myPU"
    />
    <context:component-scan base-
package="pl.kuku.myspringhelloworld" />
</beans>
```

I run the program – it works.

5.12. @PersistenceUnit and @PersistenceContext

Spring understands JPA's annotations *@PersistenceUnit and @PersistenceContext*. So I can change DBDataDisplayer this way:

```
package pl.kuku.myspringhelloworld;
import java.util.List;
import javax.annotation.PostConstruct;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
import org.springframework.stereotype.Component;
import pl.kuku.myspringhelloworld.entities.Customer;
@Component
public class DBDataDisplayer {
    @PersistenceUnit
    private EntityManagerFactory emf;
    @PostConstruct
    public void doWhatMustBeDone() {
        EntityManager em = emf.createEntityManager();
        List<Customer> result = em.createQuery("select c from
Customer c").getResultList();
        for (Customer c : result) {
            System.out.println(c.getName());
        }
    }
```

```
}
```

I run the program, it works.

I can also insert persistence context (it is, entity manager). I edit DBDataDisplayer:

```
package pl.kuku.myspringhelloworld;
import java.util.List;
import javax.annotation.PostConstruct;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.springframework.stereotype.Component;
import pl.kuku.myspringhelloworld.entities.Customer;
@Component
public class DBDataDisplayer {
    @PersistenceContext
    private EntityManager em;
    @PostConstruct
    public void doWhatMustBeDone() {
        List<Customer> result = em.createQuery("select c from
Customer c").getResultList();
        for (Customer c : result) {
            System.out.println(c.getName());
        }
    }
}
```

I run the program. It works.

Hm, but is it a good idea to have entity manager injected? As we already know, all beans' objects that Spring creates are singletons – Spring creates them on application context start and then keeps a single instance of them. So if two threads simultaneously run some method on my class which has entity manager injected, they will – I may suspect – act on the same entity manager, but entity manager is not thread safe.

Fortunately, what we have injected is not just an entity manager. I edit DBDataDisplayer:

```
package pl.kuku.myspringhelloworld;
import java.util.List;
import javax.annotation.PostConstruct;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.springframework.stereotype.Component;
import pl.kuku.myspringhelloworld.entities.Customer;
@Component
public class DBDataDisplayer {
    @PersistenceContext
    private EntityManager em;
    @PostConstruct
    public void doWhatMustBeDone() {
        System.out.println("entity manager classname: " +
em.getClass().getName());
        List<Customer> result = em.createQuery("select c from
Customer c").getResultList();
        for (Customer c : result) {
            System.out.println(c.getName());
        }
    }
}
```

I run the program. It prints out:

```
entity manager classname: com.sun.proxy.$Proxy7
Jumbo Eagle Corp
New Enterprises
Wren Computers
(...)
```

As we can see, what we have injected in the *em* attribute is some proxy. It is a proxy that keeps actual entity manager in thread local and delegates all method to it – so if many threads access this proxy, each thread has and uses its own entity manager.

TODO: weaving.

6. Spring MVC

The program I write in this chapter and in following chapters is in the project *MySpringMVCProject*.

6.1. I install and configure Tomcat

I install Tomcat the usual way.

I edit its *apache-tomcat-8.0.21/conf/tomcat-users.xml* file, I add a user there:

```
<tomcat-users (...)>
     <role rolename="manager-gui"/>
     <role rolename="manager-script"/>
     <user username="admin" password="bigaiM4o" roles="manager-
gui,manager-script" />
</tomcat-users>
```

I start Tomcat the usual way.

6.2. I create a new project and deploy it in Tomcat

In NetBeans I create a new project of type *maven -> java application*. I call it *MySpringMVCProject*.

Now I prepare this application do be deployed to Tomcat.

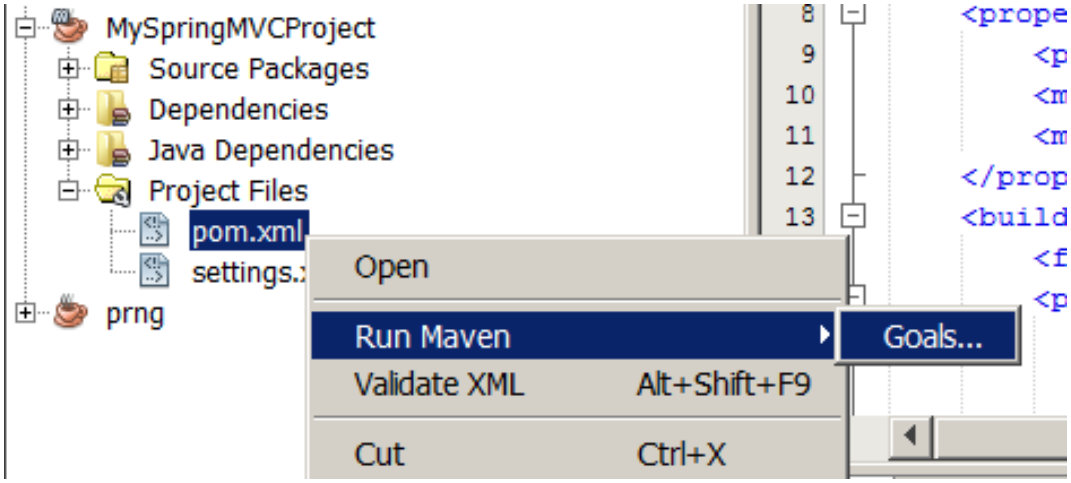
I configure maven so it knows Tomcat's username and password. I am under Windows now, so I edit maven's configuration file *c:\Users\admin\.m2\settings.xml*. I write there:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                      http://maven.apache.org/xsd/settings-
1.0.0.xsd">
     <servers>
          <server>
               <id>TomcatServer</id>
               <username>admin</username>
               <password>bigaiM4o</password>
          </server>
     </servers>
</settings>
```

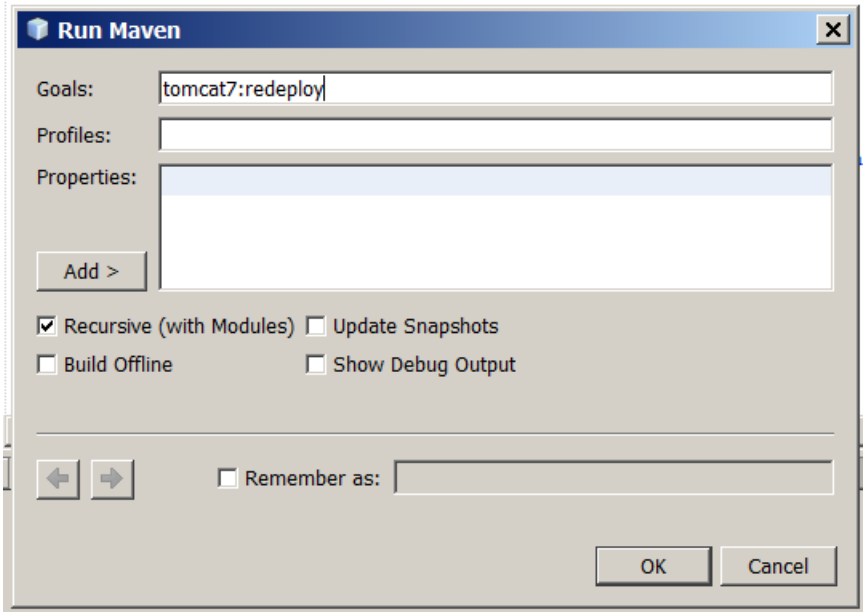
Now I edit my project's pom file. I add there configuration for the *tomcat7-maven-plugin* plugin:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  (...)
  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
        <configuration>
          <url>http://localhost:8080/manager/text</url>
          <server>TomcatServer</server>
          <path>/ink</path>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Now I right click on *pom.xml*, do *run maven -> goals*:



I run *tomcat7:deploy* (or *tomcat7:redeploy*, if I deployed this application before):



I get such warnings:

```
--- maven-jar-plugin:2.3.2:jar (default-jar) @ MySpringMVCProject
---
JAR will be empty - no content was marked for inclusion!
Building jar: C:\Users\admin\Documents\NetBeansProjects\
MySpringMVCProject\target\MySpringMVCProject.jar

<<< tomcat7-maven-plugin:2.2:redeploy (default-cli) @
MySpringMVCProject <<<

--- tomcat7-maven-plugin:2.2:redeploy (default-cli) @
MySpringMVCProject ---
Skipping non-war project
```

It is because project's *pom.xml* contains this:

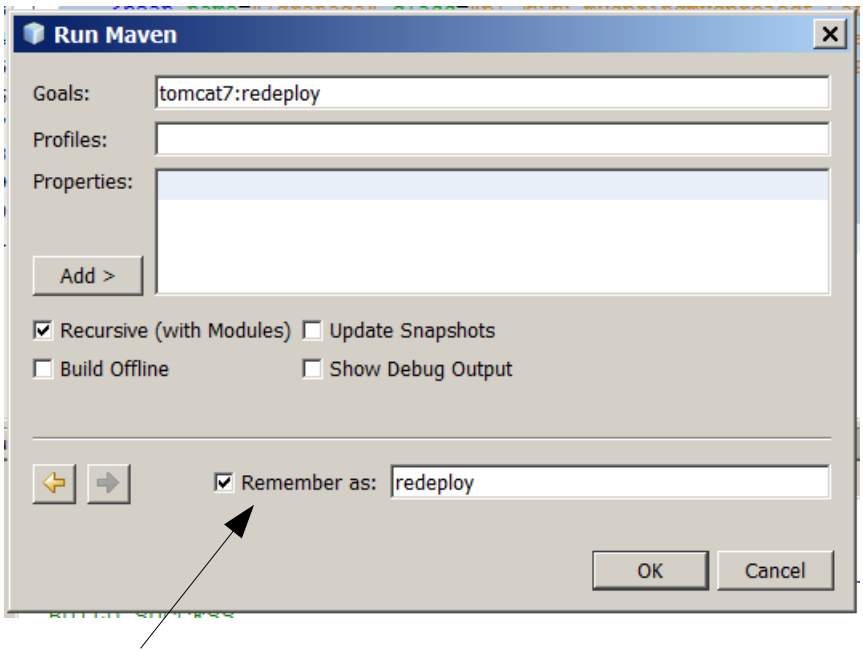
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>pl.kuku</groupId>
    <artifactId>MySpringMVCProject</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
```

So, my project's gets packed to jar. I edit project's *pom.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>pl.kuku</groupId>
    <artifactId>MySpringMVCProject</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>
```

I again deploy my application (by running *tomcat7:redeploy* goal).

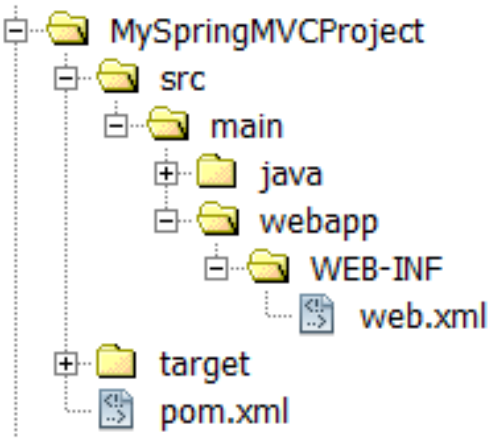
I will use this goal quite often, so it may make sense to make netbeans remember this goal:



Now, after I redeployed the application, I get this error message:

```
Failed to execute goal org.apache.maven.plugins:maven-war-
plugin:2.1.1:war (default-war) on project MySpringMVCProject:
Error assembling WAR: webxml attribute is required (or pre-
existing WEB-INF/web.xml if executing in update mode) -> [Help 1]
```

It is because my project does not contain *web.xml*. In eclipse I go to the *files* tab and there I create *src/main/webapp/WEB-INF/web.xml* file:



In this *web.xml* file I write:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">
</web-app>
```

I again deploy my application (by running *tomcat7:redeploy* goal).

Now the application gets deployed. I can see it in Eclipse's output window:

```
--- tomcat7-maven-plugin:2.2:redeploy (default-cli) @
MySpringMVCProject ---
Deploying war to http://localhost:8080/ink
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for
further details.
Uploading: http://localhost:8080/manager/text/deploy?path=
%2Fink&update=true
Uploaded: http://localhost:8080/manager/text/deploy?path=
%2Fink&update=true (3 KB at 111.9 KB/sec)
tomcatManager status code:200, ReasonPhrase:OK
OK - Deployed application at context path /ink
```

I also take a look at Tomcat's *apache-tomcat-8.0.21/logs/catalina.2015-05-06.log* file:

```
06-May-2015 19:45:46.764 INFO [http-nio-8080-exec-130]
org.apache.catalina.startup.HostConfig.undeploy Undeploying
context [/ink]
06-May-2015 19:45:47.167 INFO [http-nio-8080-exec-130]
org.springframework.context.support.AbstractApplicationContext.doC
lose Closing WebApplicationContext for namespace 'dispatcher-
servlet': startup date [Tue May 05 20:21:37 CEST 2015]; root of
context hierarchy
06-May-2015 19:45:49.249 INFO [http-nio-8080-exec-130]
org.apache.catalina.startup.HostConfig.deployWAR Deploying web
application archive D:\programfiles\apache-tomcat-8.0.21\webapps\
ink.war
```

```
06-May-2015 19:45:51.225 INFO [http-nio-8080-exec-130]
org.apache.catalina.startup.HostConfig.deployWAR Deployment of web
application archive D:\programfiles\apache-tomcat-8.0.21\webapps\
ink.war has finished in 1,940 ms
```

Great! It means my application is deployed.

6.3. dispatcher servlet

The main thing in Spring MVC is a dispatcher servlet. It is a servlet written by the authors of Spring – it is the class *org.springframework.web.servlet.DispatcherServlet*. We configure the servlet containers so that some requests are passed to it.

Then we need **handlers**. A handler is an object which can produce a response to the request. It can be an object of any type, containing any methods, receiving any arguments and producing result of any type. When the **dispatcher servlet** gets some HTTP request, it consults all registered **handler mappings** and asks them if they can give him a handler for this request. One of handler mappings produces handler and now dispatcher servlet has to forward the HTTP request to this handler, so the handler produces the result and the dispatcher servlet can send this result back to the client. Ok, actually the previous sentence was an oversimplification. Actually, handler mapping does not produce handler – it produces a **handler execution chain**, which is a container containing one handler and any number of **interceptors**. However, in first, simple experiments we will not care about interceptors.

But the handler can be any object with any method, so how can dispatcher servlet call the handler? And that's why in Spring MVC there are **handler adapters**. Dispatcher servlet calls all registered handler adapters and asks them if they can help him to call this handler. One of handler adapters – probably written by the same author that wrote the handler mapping that produced the handler – answers that it can call this handler. So the dispatcher servlet calls this handler adapter and passes it request, response and handler objects. In this moment handler adapter can do one of two things. Handler adapter has a response object (it got it from the dispatcher servlet), so it can write the response directly to this response object. But there is also the other thing that handler adapter can do (and this second thing is more typical). Handler adapter can return (to the dispatcher servlet, which called it) a **ModelAndView** object. This ModelAndView is just a thin container containing two objects: a **model**, which contains data produced by the handler, and a either a **view** (which is an object that can render the data that are in the model) or the name of the view (that name will be later used to find the view). The model has to be an object of type ModelMap, which is just a map whose keys are strings and values are any objects. The view has to be an object of type View. The view name has to be a String. So each ModelView object has two fields: one contains a Model and the other contains either a View or a String being a logical name of some view. If now you wonder what does it mean *a logical name of some view*, hold on – I will explain it soon.

When the dispatcher servlet has the model and the view produced by the handler adapter from the result produced by the handler, it has to call the view's method *handle* and pass it the model, the request object and the response object – so that the view can produce a text representation of the model and write it to the response object. So the dispatcher servlet needs a view. If the ModelAndView object returned by the handler adapter contains

the view object, the dispatcher servlet has the view object. If the ModelAndView object returned by the handler adapter does not contain the view object but only the logical name of the view, the dispatcher servlet consults all registered **view resolver** if any of them can give the view for this logical view name.

As we can see, dispatcher servlet processes the request using several objects: handler mapping, handler, handler adapter, model, view and view adapter. Some of that objects – handler and view – do not depend on Spring and can be objects of any type. Some of objects – handler mapping, handler adapter, model and view adapter – depend on Spring and act as a bridge between Spring independent beans (handler and view) and Spring. Such design gives great flexibility. Thanks to it we can use any views and handlers with Spring.

So, if I want that the dispatcher servlet answers to some URLs, I have to have in application context four beans: handler mapping, handler, handler adapter and view adapter. In typical situations we write only handlers, and we use handler mapping, handler adapter and view resolver that authors of Spring MVC prepared for us. However, it may be a good idea to begin with writing my own very simple handler mapping, handler adapter and view resolver.

6.4. I do the basic configuration of the dispatcher servlet

Now I edit the *web.xml* file so that all requests to any URL are served by Spring's dispatcher servlet:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

I redeploy it (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/* . I get 404 error. I take a look at Tomcat's *apache-tomcat-8.0.21/logs/localhost.2015-05-08.log* log file, where I can see this:

```
08-May-2015 23:30:01.262 INFO [http-nio-8080-exec-139]
org.apache.catalina.core.ApplicationContext.log Marking servlet
dispatcher as unavailable
```

```
08-May-2015 23:30:01.262 SEVERE [http-nio-8080-exec-139]
org.apache.catalina.core.StandardContext.loadOnStartup Servlet
[dispatcher] in web application [/ink] threw load() exception
 java.lang.ClassNotFoundException:
org.springframework.web.servlet.DispatcherServlet
(...)
```

So it does not work, because my war does not contain Spring's *DispatcherServlet* class. I edit Tomcat's *pom.xml*, adding there a dependency on Spring MVC:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>pl.kuku</groupId>
  <artifactId>MySpringMVCProject</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>
  <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>
  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
        <configuration>
          <url>http://localhost:8080/manager/text</url>
          <server>TomcatServer</server>
          <path>/ink</path>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>4.0.1.RELEASE</version>
    </dependency>
  </dependencies>
</project>
```

I redeploy it (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/* . I get 500 error with the message:

```
(...)
java.io.FileNotFoundException: Could not open ServletContext
resource [/WEB-INF/dispatcher-servlet.xml]
(...)
```

It is because this servlet needs the configuration file */WEB-INF/dispatcher-servlet.xml*. Where the name of this config file comes from? Let's do an experiment. I edit *pom.xml*:

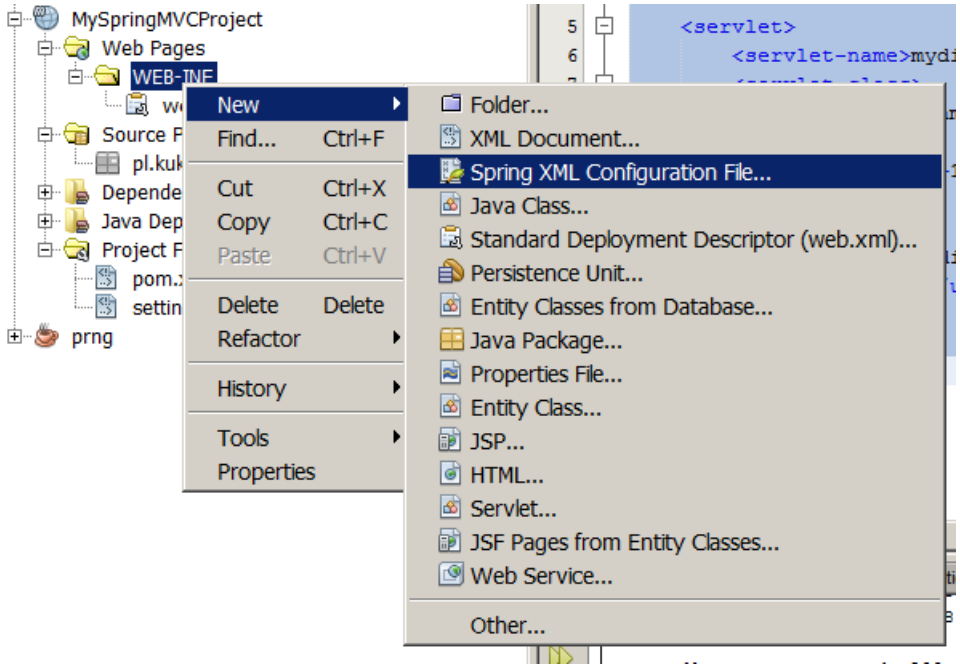
```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">
    <servlet>
        <servlet-name>mydispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>mydispatcher</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

I redeploy it (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/* . I get 500 error with the message:

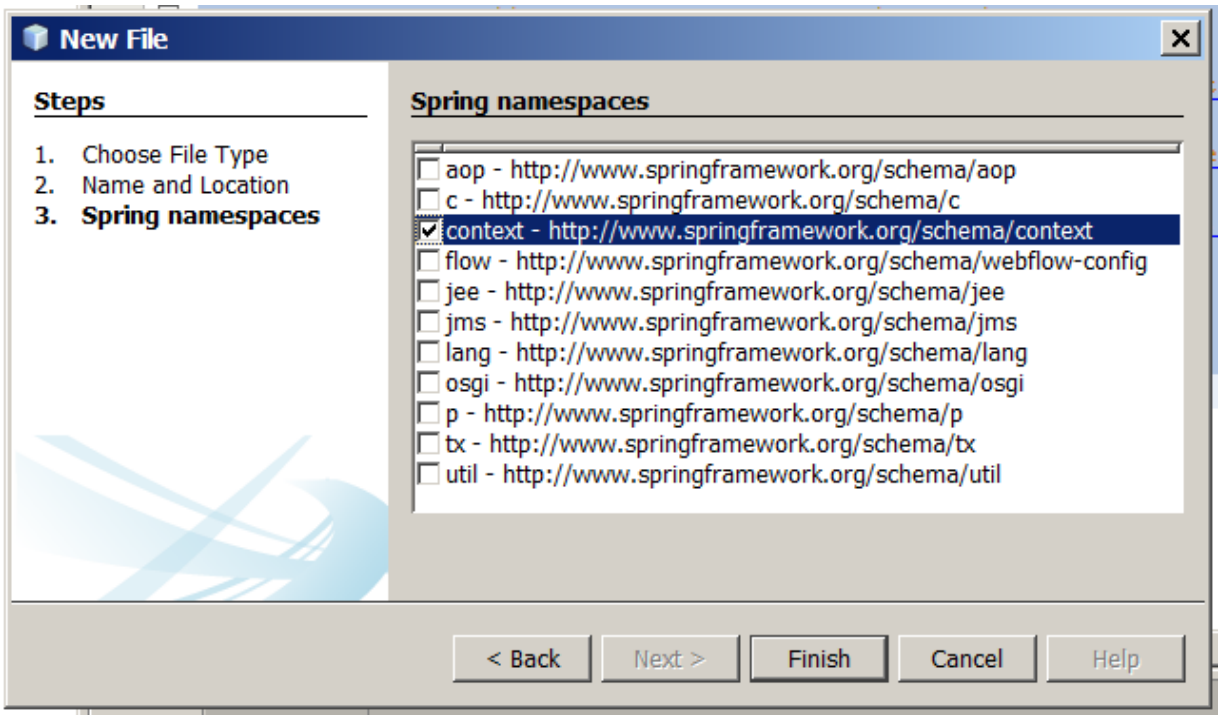
```
(...)
java.io.FileNotFoundException: Could not open ServletContext
resource [/WEB-INF/mydispatcher-servlet.xml]
(...)
```

As we can see, the name of the configuration file is based on the name of servlet's configuration.

So now I create this configuration file – */WEB-INF/mydispatcher-servlet.xml* . I right click on my project's *WEB-INF* and choose *new -> spring XML configuration file*:



I call the file *mydispatcher-servlet.xml*. I choose to use only one namespace – the *context* namespace.



I leave this config file as it is generated by Netbeans, I don't write anything in it:

```
<?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-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/* . I get the 404 error. When I take a look at logs, in *apache-tomcat-8.0.21/logs/host-manager.2015-05-04.log* I can see:

```
11-May-2015 10:35:24.869 WARNING [http-nio-8080-exec-6]
org.springframework.web.servlet.DispatcherServlet.noHandlerFound
No mapping found for HTTP request with URI [/ink/] in
DispatcherServlet with name 'mydispatcher'
```

It's normal – dispatcher servlet has not found mapping for that URL. I will take care of it later. Now I will check if Spring will create beans if I define them in *mydispatcher-servlet.xml*.

So, I create a class called *pl.kuku.myspringmvcproject.Dog*:

```
package pl.kuku.myspringmvcproject;
public class Dog {
    public Dog() {
        throw new RuntimeException("hau hau!");
    }
}
```

In */WEB-INF/mydispatcher-servlet.xml* I write:

```
<?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-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean class="pl.kuku.myspringmvcproject.Dog"
lazy-init="false"></bean>
</beans>
```

I redeploy the application. When I take a look at logs, in *apache-tomcat-8.0.21/logs/host-manager.2015-05-04.log* I can see:

```
11-May-2015 11:08:23.422 SEVERE [http-nio-8080-exec-9]
org.springframework.web.servlet.FrameworkServlet.initServletBean
Context initialization failed
 org.springframework.beans.factory.BeanCreationException: Error
creating bean with name 'pl.kuku.myspringmvcproject.Dog#0' defined
in ServletContext resource [/WEB-INF/mydispatcher-servlet.xml]:
```

```
Instantiation of bean failed; nested exception is
org.springframework.beans.BeanInstantiationException: Could not
instantiate bean class [pl.kuku.myspringmvcproject.Dog]:
Constructor threw exception; nested exception is
java.lang.RuntimeException: hau hau!
```

It means that Spring has instantiated my bean.

6.5. I configure the dispatcher servlet so that some URLs are handled by some handlers and produce some HTTP response

Now I will configure the dispatcher servlet so that some URLs are handled by some handlers.

6.5.1. I write a hello world handler mapper

First I will write a basic handler mapper, which does nothing, but only throws a runtime exception – so that I can see that the dispatcher servlet really calls it.

So, I write such class:

```
package pl.kuku.myspringmvcproject;
import javax.servlet.http.HttpServletRequest;
import org.springframework.web.servlet.HandlerExecutionChain;
import org.springframework.web.servlet.HandlerMapping;
public class VerySimpleHandlerMapping implements HandlerMapping {
    @Override
    public HandlerExecutionChain getHandler(HttpServletRequest
request) throws Exception {
        throw new RuntimeException("Hello! I'm your handler
mapping!");
    }
}
```

If I want that the dispatcher servlet uses it, I just have to define its bean in the context of dispatcher servlet (in */WEB-INF/mydispatcher-servlet.xml*):

```
<?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-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean
class="pl.kuku.myspringmvcproject.VerySimpleHandlerMapping" />
```

```
</beans>
```

However, the type *ServletHttpRequest* – which my handler adapter will get from dispatcher servlet and then will pass to my handler (implementing my interface) – is not defined in Spring, it is a part of servlet API. So in order to use it in my project I have to add some implementation of servlet API to maven dependencies of my project.

So I edit my project's *pom.xml* adding there a dependency on *org.jboss.spec.javax.servlet:jboss-servlet-api_3.0_spec* :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>pl.kuku</groupId>
  <artifactId>MySpringMVCProject</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>
  <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>
  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
        <configuration>
          <url>http://localhost:8080/manager/text</url>
          <server>TomcatServer</server>
          <path>/ink</path>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>4.0.1.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.jboss.spec.javax.servlet</groupId>
      <artifactId>jboss-servlet-api_3.0_spec</artifactId>
      <version>1.0.2.Final</version>
    </dependency>
  </dependencies>
</project>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/* . I get such error:

```
org.springframework.web.util.NestedServletException: Request
processing failed; nested exception is java.lang.RuntimeException:
Hello! I'm your handler mapping!
```

It means my handler mapping works.

6.5.2. I write some real handler and its handler mapping and handler adapter

When I write my own handler mapping and my own handler adapter, I can freely choose how will my handlers look like – what methods they will have, what arguments they will receive and what type of answer they will produce. So I choose that – for this experiment – my handlers will have a method called *doYourDuty*, receiving (as an argument) an *ServletHttpRequest* and returning just a String with a text of the response.

So I create an interface called MyTypeOfHandler:

```
package pl.kuku.myspringmvcproject;
import javax.servlet.http.HttpServletRequest;
public interface MyTypeOfHandler {
    String doYourDuty(HttpServletRequest req);
}
```

I also create a HandlerMapping called MyHandlerMapping. It will be very simple. It will have one handler and no matter what the URL is, it will always return this one handler:

```
package pl.kuku.myspringmvcproject;

import javax.servlet.http.HttpServletRequest;
import org.springframework.web.servlet.HandlerExecutionChain;
import org.springframework.web.servlet.HandlerMapping;

public class MyHandlerMapping implements HandlerMapping {
    private MyTypeOfHandler handler;

    public void setHandler(MyTypeOfHandler handler) {
        this.handler = handler;
    }

    @Override
    public HandlerExecutionChain getHandler(HttpServletRequest
request) throws Exception {
        return new HandlerExecutionChain(handler);
    }
}
```

Now I create a handler of a type *MyTypeOfHandler*, I call it *MySimpleHandler*. It is simple – all it does is to return the string *hello world*:

```
package pl.kuku.myspringmvcproject;
import javax.servlet.http.HttpServletRequest;
public class MySimpleHandler implements MyTypeOfHandler {
    @Override
    public String doYourDuty(HttpServletRequest req) {
        return "hello world";
    }
}
```

Now I configure the Spring so that it uses my handler mapper which produces my handler. In */WEB-INF/mydispatcher-servlet.xml* I write:

```
<?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-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="myhandler"
class="pl.kuku.myspringmvcproject.MySimpleHandler" />
    <bean class="pl.kuku.myspringmvcproject.MyHandlerMapping">
        <property name="handler" ref="myhandler" />
    </bean>
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/* . I get such error:

```
javax.servlet.ServletException: No adapter for handler
[pl.kuku.myspringmvcproject.MySimpleHandler@74cc94]: The
DispatcherServlet configuration needs to include a HandlerAdapter
that supports this handler
```

As we can see, I lack the handler adapter which can run handlers of type *MyTypeOfHandler*. I write such handler adapter, I call it *MyVerySimpleHandlerAdapter*:

```
package pl.kuku.myspringmvcproject;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerAdapter;
import org.springframework.web.servlet.ModelAndView;

public class MyVerySimpleHandlerAdapter implements HandlerAdapter
{

    @Override
    public boolean supports(Object handler) {
        return handler instanceof MyTypeOfHandler;
    }

    @Override
    public ModelAndView handle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        String result = ((MyTypeOfHandler)
handler).doYourDuty(request);
        throw new RuntimeException(result);
    }

    @Override
    public long getLastModified(HttpServletRequest request, Object
handler) {
        return -1;
    }

}
```

As we can see, it is really very simple – when it gets the result text from the handler, it throws a runtime exception with this result text.

I put the bean of type *MyVerySimpleHandlerAdapter* in the *mydispatcher-servlet.xml* file:

```
<?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-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="myhandler"
class="pl.kuku.myspringmvcproject.MySimpleHandler" />
    <bean class="pl.kuku.myspringmvcproject.MyHandlerMapping">
        <property name="handler" ref="myhandler" />
    </bean>
    <bean
class="pl.kuku.myspringmvcproject.MyVerySimpleHandlerAdapter" />
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/* . I get such error:

```
java.lang.RuntimeException: hello world
pl.kuku.myspringmvcproject.MyVerySimpleHandlerAdapter.handle(MyVer
ySimpleHandlerAdapter.java:18)
```

It means that my handler and handler adapter work.

Now I will make my handler adapter really work. I mean, I will make it actually send some pretty text to the browser and not just throw an exception.

I could make the handler adapter write directly to the *HttpServletResponse* object, which the *handle* method receives from the dispatcher servlet. But it's not typical, normal handler adapters don't work that way. So instead I will make my handler adapter return ModelAndView. The model will contain the text that the handler produced and the View will write the model's text to response. So I create the handler adapter called *MySecondHandlerAdapter*:

```
package pl.kuku.myspringmvcproject;

import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerAdapter;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.View;

```

```
public class MySecondHandlerAdapter implements HandlerAdapter {
    @Override
    public boolean supports(Object handler) {
        return handler instanceof MyTypeOfHandler;
    }

    @Override
    public ModelAndView handle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        String result = ((MyTypeOfHandler)
handler).doYourDuty(request);
        Map<String, String> model = new HashMap<>();
        model.put("result", result);
        View view = new View() {
            @Override
            public String getContentType() {
                return "text/html";
            }
            @Override
            public void render(Map<String, ?> model,
HttpServletRequest request, HttpServletResponse response) throws
Exception {
                response.setContentType("text/html");
                PrintWriter writer = response.getWriter();
                writer.write((String) model.get("result"));
                writer.close();
            }
        };
        return new ModelAndView(view, model);
    }

    @Override
    public long getLastModified(HttpServletRequest request, Object
handler) {
        return -1;
    }

}
```

I edit *mydispatcher-servlet.xml* to use this handler adapter instead of the previous one:

```
<?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-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="myhandler"
class="pl.kuku.myspringmvcproject.MySimpleHandler" />
    <bean class="pl.kuku.myspringmvcproject.MyHandlerMapping">
        <property name="handler" ref="myhandler" />
    </bean>
    <bean
class="pl.kuku.myspringmvcproject.MySecondHandlerAdapter" />
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/* . This time I don't gent an error but just the text:

```
hello world
```

However, it's still not the way normal handler adapters work. Normal adapters don't produce the view. Normal adapters just return the logical view name and it's the job of a view resolver to produce the view for a given logical view name.

So I create the class called *MySimpleView* and I copy there some code from *MySecondHandlerAdapter*:

```
package pl.kuku.myspringmvcproject;

import java.io.PrintWriter;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.View;

public class MySimpleView implements View {

    @Override
    public String getContentType() {
        return "text/html";
    }

    @Override
    public void render(Map<String, ?> model, HttpServletRequest
request, HttpServletResponse response) throws Exception {
        response.setContentType("text/html");
        PrintWriter writer = response.getWriter();
        writer.write((String) model.get("result"));
        writer.close();
    }

}
```

I also create a copy of *MySecondHandlerAdapter* called *MyThirdHandlerAdapter*. I edit it so that it does not put the view itself into the ModelAndView but only the view's logical name. That name can be any string, so I make it *bialystok*:

```
package pl.kuku.myspringmvcproject;

import java.util.HashMap;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerAdapter;
import org.springframework.web.servlet.ModelAndView;

public class MyThirdHandlerAdapter implements HandlerAdapter {

    @Override
    public boolean supports(Object handler) {
        return handler instanceof MyTypeOfHandler;
    }

    @Override
    public ModelAndView handle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        String result = ((MyTypeOfHandler)
handler).doYourDuty(request);
        Map<String, String> model = new HashMap<>();
        model.put("result", result);
        return new ModelAndView("bialystok", model);
    }

    @Override
    public long getLastModified(HttpServletRequest request, Object
handler) {
        return -1;
    }

}
```

I also create a view resolver – the class called *MyViewResolver*:

```
package pl.kuku.myspringmvcproject;

import java.util.Locale;
import org.springframework.web.servlet.View;
import org.springframework.web.servlet.ViewResolver;

public class MyViewResolver implements ViewResolver {

    @Override
    public View resolveViewName(String viewName, Locale locale)
throws Exception {
        return viewName.equals("bialystok") ? new MySimpleView() :
null;
    }

}
```

I edit *mydispatcher-servlet.xml* to use the new handler adapter and the new view resolver:

```
<?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-4.0.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.0.xsd
">
    <bean id="myhandler"
class="pl.kuku.myspringmvcproject.MySimpleHandler" />
    <bean class="pl.kuku.myspringmvcproject.MyHandlerMapping">
        <property name="handler" ref="myhandler" />
    </bean>
    <bean class="pl.kuku.myspringmvcproject.MyThirdHandlerAdapter" />
    <bean class="pl.kuku.myspringmvcproject.MyViewResolver" />
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/* . I see the text:

```
hello world
```

It means everything works.

6.5.3. I use handler adapter and handler interface written by authors of Spring MVC

In practice we usually don't write all those classes by ourselves. Of course, we have to write our own handlers – they are the controllers of our application – but we can use handler interfaces, handler mappers and handler adapters that the authors of Spring MVC have created.

6.5.3.1. I use a handler interface called *Controller*

There is a handler interface called *Controller*. There is also a handler adapter which can use handlers of type *Controller* – it is called *SimpleControllerHandlerAdapter*. The *Controller* interface is very simple, similar to our *MyTypeOfHandler*. It has just one method, called *handleRequest*, which expects to receive *HttpServletRequest* and *HttpServletResponse* methods. This method must return ModelAndView.

When talking about classes implementing the Controller interface and their instances, I will call them just *controllers*.

Spring MVC authors decided that there are some basic things we probably want to do in every controller. For instance, to synchronize on session (so our controller can use session in multiple threads) or to take care about caching. Spring MVC authors have created a class implementing all that functionality. This is the abstract class called *AbstractController.* This class uses *template method* design pattern – it has an abstract *handleRequestInternal* method that we have to override.

So I create a simple controller extending AbstractController – class *pl.kuku.myspringmvcproject.CatController*:

```
package pl.kuku.myspringmvcproject;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
public class CatController extends AbstractController {
    @Override
    protected ModelAndView
handleRequestInternal(HttpServletRequest hsr, HttpServletResponse
hsr1) throws Exception {
        throw new RuntimeException("miau!");
    }
}
```

As we can see, so far it is very simple – it does not return any model or view, it just throws an exception.

6.5.3.2. I use SimpleUrlHandlerMapping to map some URLs to the CatController

In order to map some URLs to my controller I have to use some handler mapping. There are several ready made handler mappings that I can use. One simple handler mapping is *SimpleUrlHandlerMapping*. It is a class whose objects keep a dictionary mapping URL patterns to objects or bean names. When this handler mapping is asked to provide a handler for a given URL, it compares that URL with patterns and returns the object of the

pattern that matched. In the pattern we can use the syntax of the AntPathMatcher (*http://docs.spring.io/spring-framework/docs/2.5.6/api/org/springframework/util/AntPathMatcher.html*).

So in *mydispatcher-servlet.xml* I configure three bean definitions: a definition of *CatController*, a definition of *SimpleUrlHandlerMapping* and a definition of *org.springframework.web.servlet.mvc.SimpleControllerHandler* (this is the handler adapter which can call controllers).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMap
ping">
        <property name="urlMap">
            <map>
                <entry key="/saragossa">
                    <bean
class="pl.kuku.myspringmvcproject.CatController" />
                </entry>
            </map>
        </property>
    </bean>
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandler
Adapter" />
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/saragossa* . I get a runtime error:

```
java.lang.RuntimeException: miau!
pl.kuku.myspringmvcproject.CatController.handleRequestInternal(Cat
Controller.java:12)
```

It means that my controller was found and used.

I try if I really can put a bean name – and not a handler itself – in the configuration of *SimpleUrlHandlerMapping*. I change *mydispatcher-servlet.xml* this way:

```
<bean id="cat" class="pl.kuku.myspringmvcproject.CatController" />
<bean
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMap
ping">
    <property name="urlMap">
        <map>
            <entry key="/saragossa">
                <value>cat</value>
            </entry>
        </map>
    </property>
</bean>
<bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandler
Adapter" />
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/saragossa* . I get the same runtime error as before.

I could also configure *SimpleUrlHandlerMapping* by setting its *mappings* property instead of *urlMap* property. The difference between those two properties is that the *mappings* property is a property of type *java.util.Properties*, which in practice means that it is a little bit less to write to set the *mappings* property (FIXME: czy to jedyna praktyczna różnica między nimi?).

Now I improve the *CatController* so that it returns some data and some view name:

```
package pl.kuku.myspringmvcproject;

import java.util.HashMap;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class CatController extends AbstractController {

    @Override
    protected ModelAndView
handleRequestInternal(HttpServletRequest hsr, HttpServletResponse
hsr1) throws Exception {
        return new ModelAndView("myView", new HashMap<String,
Integer>() {{
            put("myNumber", 3 * 7);
        }});
    }
}
```

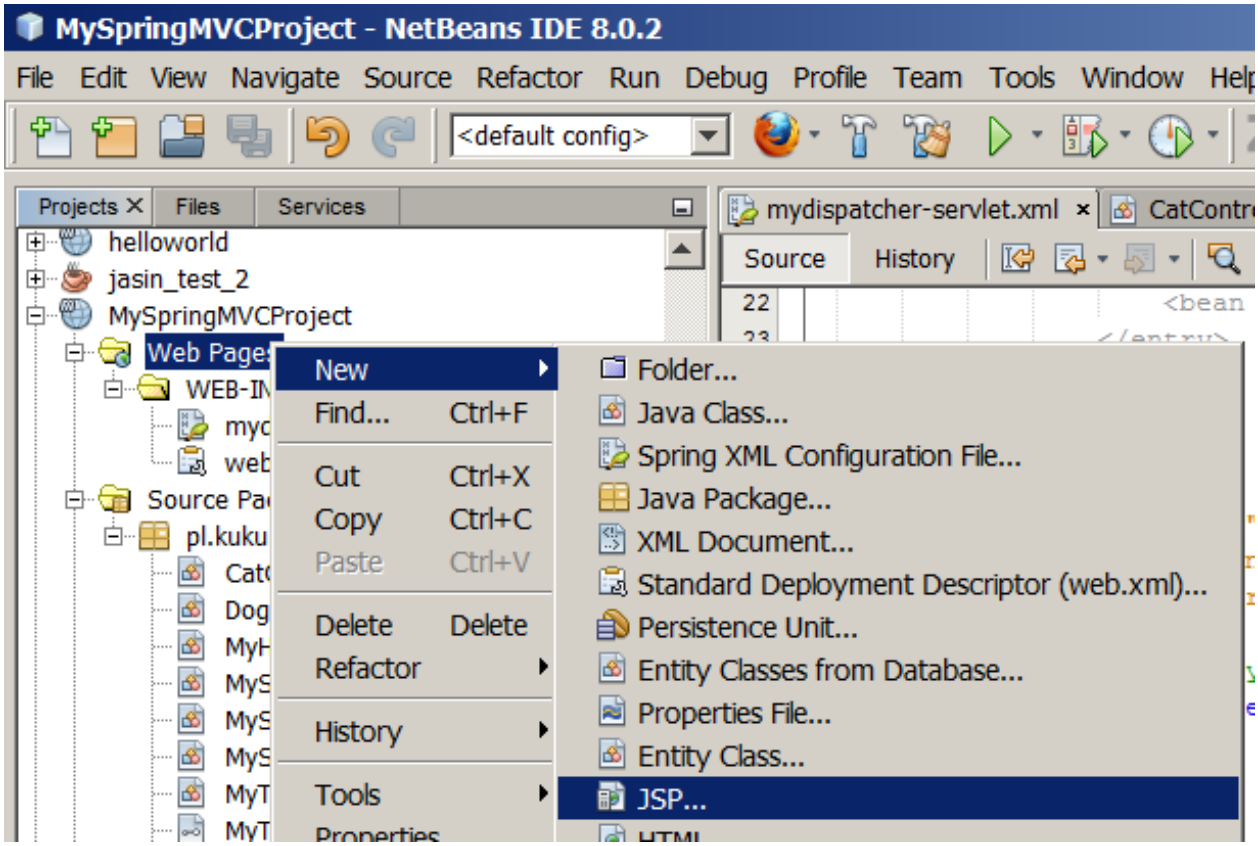
I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/saragossa* . I get a 404 error. It is (I suppose) because none of registered view resolvers could resolve *myView* to a working view.

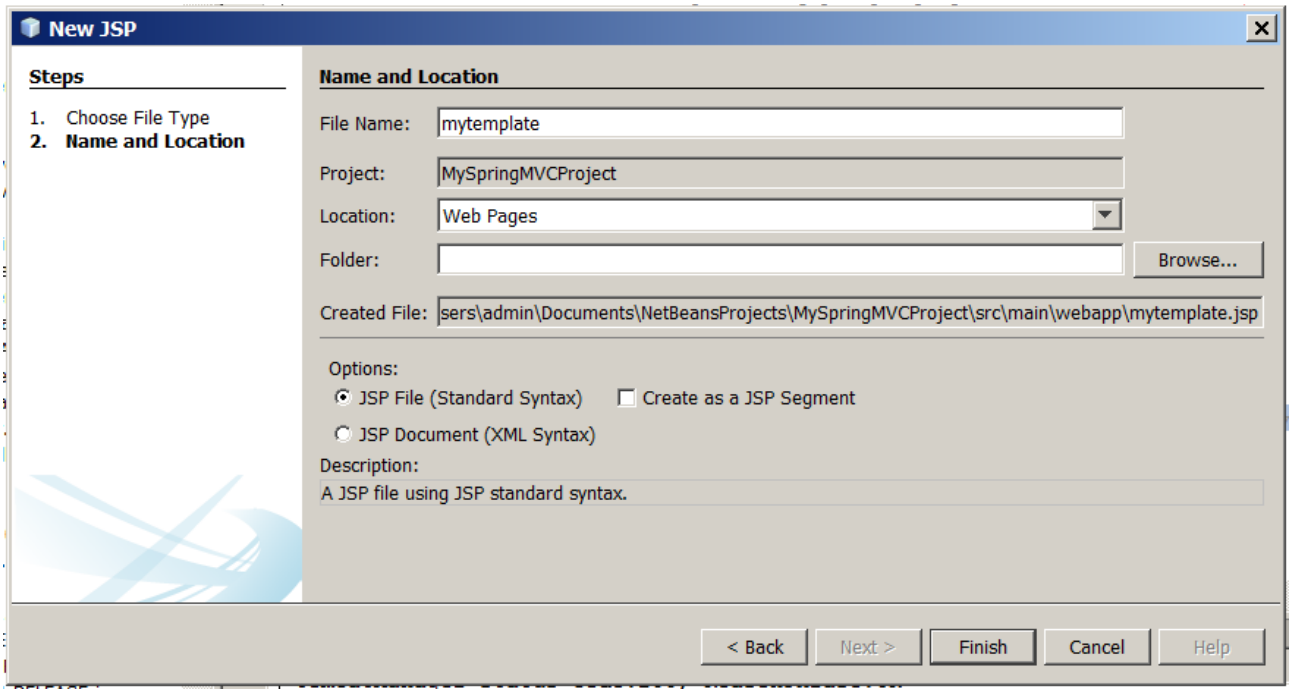
6.5.3.3. I use the *InternalResourceViewResolver* view resolver to display the model using a JSP template

I also need some view resolver to resolve this logical view name into the view itself. I will use the *org.springframework.web.servlet.view.InternalResourceViewResolver* view resolver. It is the view resolver that regardless of the logical view name always does the same. It produces the view (usually of type *InternalResourceView*, but subclasses of *InternalResourceViewResolver* can override it by overriding the *requiredViewClass* method) that takes the logical view name, prepends to it some prefix (which we can configure), appends to it some postfix (which we can also configure) and then forwards the request to this string. Actually, the *InternalResourceView* view can either forward or include. By default, it forwards, but it is possible to configure the *InternalResourceView* (using its *setAlwaysInclude* method) to make it always include. Before the forward data from model are put into the request so that the servlet to which the request is forwarded can display them (it is done by the method *AbstractView#exposeModelAsRequestAttributes*, which calls *setAttribute* on the request object).

I will use it the very typical way – to redirect everything to some JSP template.

So first I create the *mytemplate.jsp* in the Netbeans pseudodirectory called *Web Pages*:

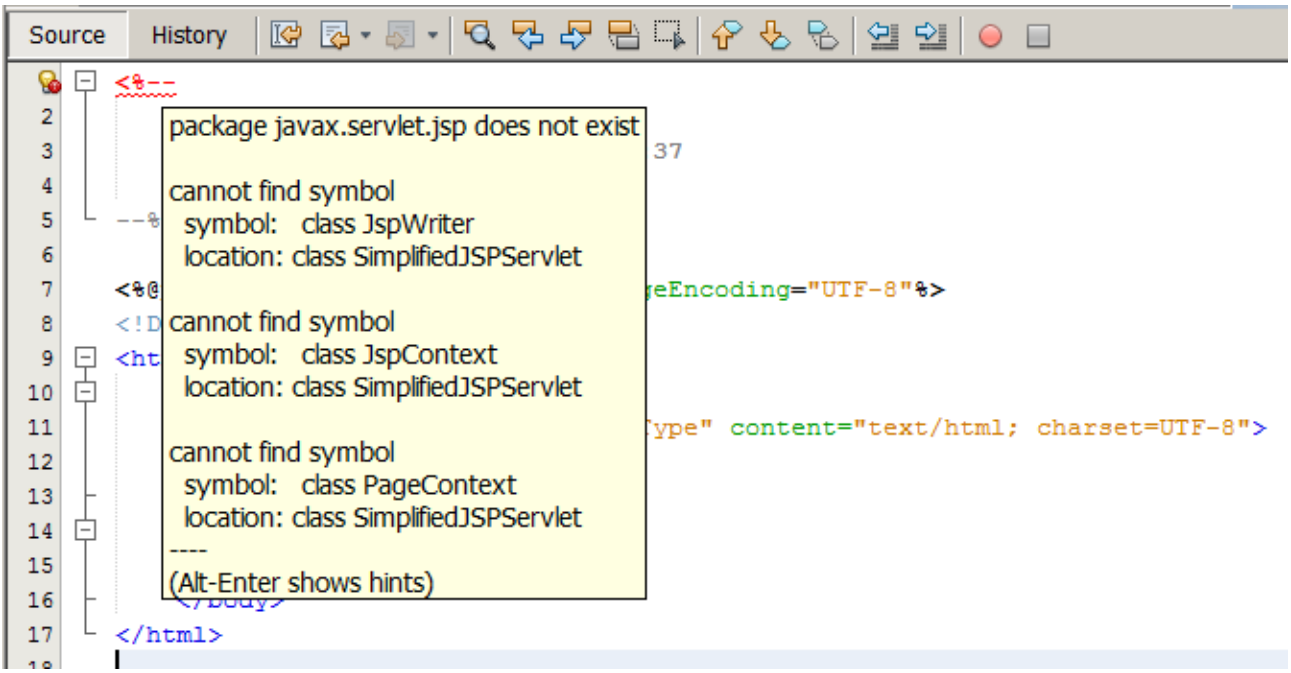




In this file netbeans automatically puts this content:

```
<%--
    Document   : mytemplate
    Created on : 2015-05-22, 18:15:37
    Author     : admin
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello World!</h1>
    </body>
</html>
```

However, at the very first line netbeans displays this error:



So I have to put in maven dependencies of my project a dependency with JSP API.

So, in the *pom.xml* of my project, in it *dependencies* section I add this:

```
<dependency>
     <groupId>javax.servlet</groupId>
     <artifactId>jsp-api</artifactId>
     <version>2.0</version>
</dependency>
```

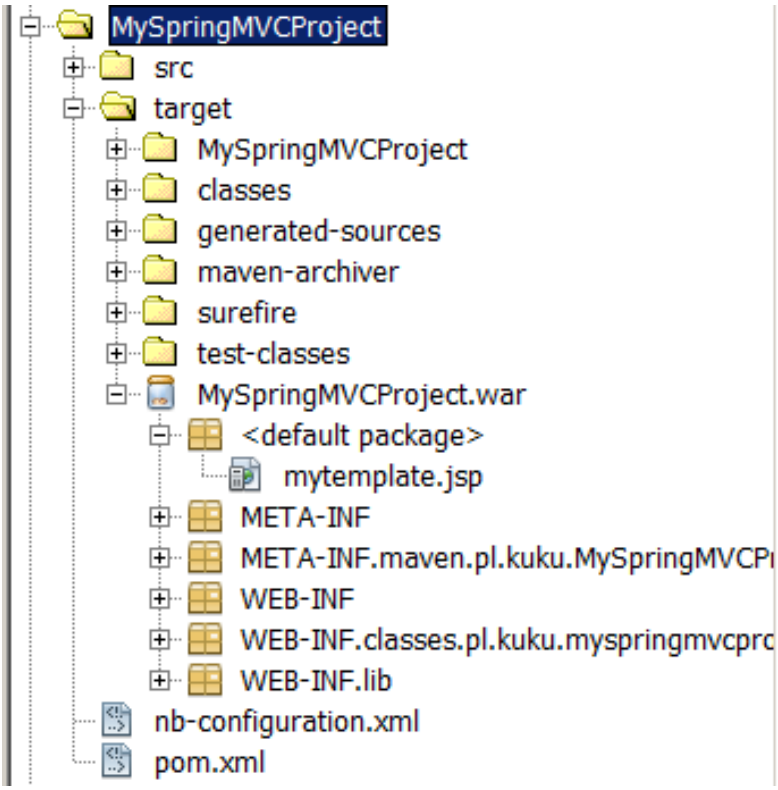
The error disappears.

Now I edit the *mytemplate.jsp* file:

```
<%--
    Document   : mytemplate
    Created on : 2015-05-22, 18:15:37
    Author     : admin
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        ${2*2*2*2}
    </body>
</html>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/mytemplate.jsp* . I get 404 error.

Let's see whether the war that is created and deployed by maven. In *netbeans* I go to *files* tab, and there in the *target* folder i take a look at the *MySpringMVCProject.war*:



Hm, so why the *mytemplate.jsp* is not found and gives a 404 error? It is because the url *http://127.0.0.1:8080/ink/mytemplate.jsp* is matched by the pattern /* in the *web.xml* file:

```
<servlet-mapping>
    <servlet-name>mydispatcher</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
```

I change the *web.xml* so that the dispatcher servlet is only the default and not catching everything[4]:

```
<servlet-mapping>
    <servlet-name>mydispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/mytemplate.jsp* . This time I get 500 error:

```
org.apache.jasper.JasperException: Unable to compile class for
JSP:
```

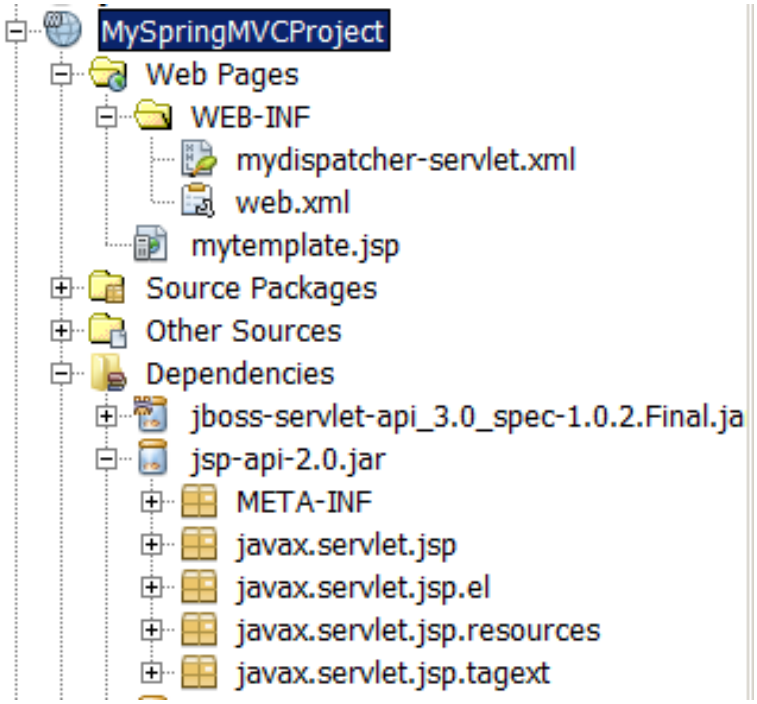
4 If you don't know what's the difference between /* and / url-pattern, read this: *https://stackoverflow.com/questions/4140448/difference-between-and-in-servlet-mapping-url-pattern?rq=1* *https://stackoverflow.com/questions/14018215/what-is-url-pattern-in-web-xml-and-how-to-configure-servlet*

```
An error occurred at line: [52] in the generated java file: [D:\
programfiles\apache-tomcat-8.0.21\work\Catalina\localhost\ink\org\
apache\jsp\mytemplate_jsp.java]
The method getJspApplicationContext(ServletContext) is undefined
for the type JspFactory
```

To understand why we have this error, we must understand what happens when the browsers requests the */ink/mytemplate.jsp* from the Tomcat. Our project does not have in its *web.xml* any pattern matching the */ink/mytemplate.jsp* path. However, Tomcat has a global *web.xml* file in the *apache-tomcat-8.0.21/conf/web.xml* file. This file says:

```
(...)
<servlet>
  <servlet-name>jsp</servlet-name>
  <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-
class>
  (...)
</servlet>
(...)
<servlet-mapping>
  <servlet-name>jsp</servlet-name>
  <url-pattern>*.jsp</url-pattern>
  <url-pattern>*.jspx</url-pattern>
</servlet-mapping>
(...)
```

It means that when the path of the resource that the browser has requested ends with *.jsp* or *.jspx* it is served by the *org.apache.jasper.servlet.JspServlet* servlet. This class is not in the *jsp-api-2.0.jar* that we added to our project when we added the JSP api to its maven dependencies:



That makes sense – the JSP API is in *javax.servlet.jsp* package and its subpackages (notice: it begins with *javax*, so it is an official package). The *org.apache.jasper.servlet* package name sounds like it is a part of some JSP implementation (notice: it begins with some domain name, so it is not an official package). So our application needs a jar with this JSP implementation. Tomcat has a jar with this JSP implementation in the *apache-tomcat-8.0.21/lib/jasper.jar* file. That makes sense – if Tomcat by default is configured to use some particular JSP implementation, it must have this implementation's jar.

The reason of this error is that the *org.apache.jasper.servlet.JspServlet* servlet tries to call the method *getJspApplicationContext* on some object of type *JspFactory*. The error message does not say that, but I know that this class qualified (full) name is *javax.servlet.jsp.JspFactory*. During the runtime our application has on its classpath two definitions of this class, because it has two jars with the JSP API. One jar is a jar that Tomcat has by default in the *apache-tomcat-8.0.21/lib/jsp-api.jar* file. The other jar is the jar that we added to our project when we added the JSP api to its maven dependencies. When we take a look at the Tomcat's jar at the definition of the *javax.servlet.jsp.JspFactory* class, we can see that it has the *getJspApplicationContext* method:

```
public abstract class javax.servlet.jsp.JspFactory {
  (...)
  public abstract javax.servlet.jsp.JspApplicationContext
getJspApplicationContext(javax.servlet.ServletContext);
```

However, if we take a look at the jar that maven added to our project at the definition of the *javax.servlet.jsp.JspFactory* class, we can see that it does not have the *getJspApplicationContext* method. I don't know why these two definitions of this class differ. Maybe those are two different versions of the JSP API (I mean, one is earlier and the other one is later)? I don't know. However, it is the cause of the problem.

So the cause of the problem is that we have two jars with JSP API. The jar that Maven has added to our project is not needed during the runtime, so I change the scope of this dependency in the *pom.xml* of my project:

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.0</version>
    <scope>provided</scope>
</dependency>
```

I redeploy the application by running Maven's *clean tomcat7:redeploy* goals (notice that we have to use the *clean* keyword here, so that the result of previous builds is deleted). I test it by visiting *http://127.0.0.1:8080/ink/mytemplate.jsp* . Now I don't get any error, but I see the result of my JSP template – the number 16.

Now I will configure the *InternalResourceViewResolver* view resolver so that when my controller ends his job and returns model and view, the view resolver will call my JSP template. The logical view name that my *CatController* controller returns is *myView*. So I rename my JSP template from *mytemplate.jsp* to *myView.jsp*. Then in *mydispatcher-servlet.xml* I configure the bean of class *InternalResourceViewResolver*:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean id="cat"
class="pl.kuku.myspringmvcproject.CatController" />
    <bean
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMap
ping">
        <property name="urlMap">
            <map>
                <entry key="/saragossa">
                    <value>cat</value>
                </entry>
            </map>
        </property>
    </bean>
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandler
Adapter" />
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewRe
solver">
        <property name="prefix" value="" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/saragossa* . I see the result of my JSP template – the number 16.

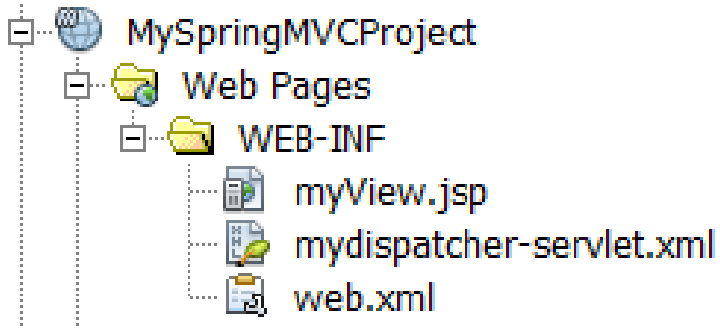
I change the template so that it displays the model produced by the controller:

```
<%--
    Document   : mytemplate
    Created on : 2015-05-23, 06:40:21
    Author     : admin
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
```

```
        <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        ${myNumber}
    </body>
</html>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/saragossa* . I see the result of the controller – the number 21.

Now I will move this JSP template to such directory that it will not be possible for a browser to request it directly. I will do it, because the official Spring documentation advices it[5]. So I move this JSP template to the *WEB-INF* folder.



5 *http://docs.spring.io/spring-framework/docs/2.5.6/api/org/springframework/web/servlet/view/InternalResourceViewResolver.html* says: BTW, it's good practice to put JSP files that just serve as views under WEB-INF, to hide them from direct access (e.g. via a manually entered URL). Only controllers will be able to access them then.

I also have to change the *InternalResourceViewResolver*'s prefix in *mydispatcher-servlet.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean id="cat"
class="pl.kuku.myspringmvcproject.CatController" />
    <bean
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMap
ping">
        <property name="urlMap">
            <map>
                <entry key="/saragossa">
                    <value>cat</value>
                </entry>
            </map>
        </property>
    </bean>
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandler
Adapter" />
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewRe
solver">
        <property name="prefix" value="WEB-INF/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/saragossa* . It still works.

6.5.3.4. I use the *BeanNameUrlHandlerMapping* handler mapping

Now I will use another handler mapping created by authors of Spring MVC – the *BeanNameUrlHandlerMapping* handler mapping. This is the mapping that is somehow similar to *SimpleUrlHandlerMapping*. It also has a mapping from URL patterns to beans. The difference is that we don't have to explicitly configure the mapping. Instead, the values of the mapping are all beans in the application context and the keys are the beans' names. Only those beans are included in the mapping whose bean names begin with a slash. In the pattern we can use the syntax of the AntPathMatcher (*http://docs.spring.io/spring-framework/docs/2.5.6/api/org/springframework/util/AntPathMatcher.html*).

So I edit the *mydispatcher-servlet.xml*. I define there a bean of class *BeanNameUrlHandlerMapping*. I also change the bean name of my controller.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean name="/granada"
class="pl.kuku.myspringmvcproject.CatController" />
    <bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerM
apping" />
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandler
Adapter" />
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewRe
solver">
        <property name="prefix" value="WEB-INF/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/granada* . It works, it shows the number 21.

Now I will try whether I can use *BeanNameUrlHandlerMapping* with handlers which are not controllers (I mean, which do not implement the *Controller* interface). I edit the *mydispatcher-servlet.xml*. I define there a bean of class *BeanNameUrlHandlerMapping* and a bean of type *pl.kuku.myspringmvcproject.MySimpleHandler* (remember? this is the simple handler I wrote several chapters before, it is an implementation of my own interface *pl.kuku.myspringmvcproject.MyTypeOfHandler*).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean name="/algeciras"
class="pl.kuku.myspringmvcproject.MySimpleHandler" />
    <bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerM
apping" />
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/algeciras* . I get an error:

```
javax.servlet.ServletException: No adapter for handler
[pl.kuku.myspringmvcproject.MySimpleHandler@139f83c]: The
DispatcherServlet configuration needs to include a HandlerAdapter
that supports this handler
```

Sure, when I use my own type of handler, I must provide my own of handler adapter. I will add (to the *mydispatcher-servlet.xml* file) the definition of a bean of type *MyVerySimpleHandlerAdapter.* Remember? This is the very simple handler adapter that can adapt handler of type *MyTypeOfHandler*. It just calls this handler and then throws the result of the handler.

So I edit *mydispatcher-servlet.xml* and I add there the handler adapter:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean name="/algeciras"
class="pl.kuku.myspringmvcproject.MySimpleHandler" />
    <bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerM
apping" />
    <bean
class="pl.kuku.myspringmvcproject.MyVerySimpleHandlerAdapter" />
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/algeciras* . It works, it throws an exception with a message *hello world*:

```
Request processing failed; nested exception is
java.lang.RuntimeException: hello world
```

It confirms that I can use *BeanNameUrlHandlerMapping* with handlers which are not controllers.

6.5.3.5. I use the *DefaultAnnotationHandlerMapping* handler mapping and the *AnnotationMethodHandlerAdapter* handler adapter

Now I will play with the *DefaultAnnotationHandlerMapping* handler mapping and the *AnnotationMethodHandlerAdapter* handler adapter.

DefaultAnnotationHandlerMapping and *AnnotationMethodHandlerAdapter* are deprecated in favor of *RequestMappingHandlerMapping* and *RequestMappingHandlerAdapter*. However, it is still useful to know these two depecated classes. It is useful to know them because they are still enabled by default in *DispatcherServlet.properties*[6] (I will talk about *DispatcherServlet.properties* later). It is also useful to know them in oder to know what are their problems and to understand how their successors – *RequestMappingHandlerMapping* and *RequestMappingHandlerAdapter* – fixed that problems.

DefaultAnnotationHandlerMapping and *AnnotationMethodHandlerAdapter* are very often used together. However, when learning about them it is a good idea to use them separately fist, because otherwise it wouldn't be easy to understand that does *DefaultAnnotationHandlerMapping* and what does *AnnotationMethodHandlerAdapter*.

6.5.3.5.1. I use the *DefaultAnnotationHandlerMapping* handler mapping

DefaultAnnotationHandlerMapping is another mapping which has a map where keys are URL patterns and values are handlers. However, we do not configure this mapping by calling some setter on the object of *DefaultAnnotationHandlerMapping* class. Instead, we

6 It was true in 2015, when I first wrote this document. Now, in 2019, it is no longer true. *DefaultAnnotationHandlerMapping* and *AnnotationMethodHandlerAdapter* are not enabled by default. Probably there is no reason to know them anymore.

annotate handlers with a *@RequestMapping* controller, and in the *value* parameter of this annotation we set the URL pattern. We can use this handler mapping with any type of handler.

In order to test it, I create a copy of *CatController*, I uncomment in it the line that throws a runtime exception (so that I don't have to think about the view) and I annotate this controller appropriately:

```
package pl.kuku.myspringmvcproject;

import java.util.HashMap;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

@RequestMapping("/cadiz")
public class CatControllerAnnotated extends AbstractController {

    @Override
    protected ModelAndView
handleRequestInternal(HttpServletRequest hsr, HttpServletResponse
hsr1) throws Exception {
        throw new RuntimeException("hello from
CatControllerAnnotated!");
    }
}
```

Now I configure application context (in file *mydispatcher-servlet.xml*). I create three beans: *CatControllerAnnotated* which will serve as a handler, *DefaultAnnotationHandlerMapping* which will map requests to it and *SimpleControllerHandlerAdapter* which knows how to execute controllers (it is, handlers of type *Controller*):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean
class="pl.kuku.myspringmvcproject.CatControllerAnnotated" />
    <bean
class="org.springframework.web.servlet.mvc.annotation.DefaultAnnot
ationHandlerMapping" />
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandler
Adapter" />
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/cadiz* . It works, it throws an exception with a message *hello from CatControllerAnnotated!*:

```
Request processing failed; nested exception is
java.lang.RuntimeException: hello from CatControllerAnnotated!
```

6.5.3.5.2. I use the *AnnotationMethodHandlerAdapter* handler adapter

The *AnnotationMethodHandlerAdapter* handler adapter is a handler adapter which can execute handlers which can be of any type, which do not have to implement any special interface, but which must have some methods annotated with an *org.springframework.web.bind.annotation.RequestMapping* annotation – the same same annotation that we use on classes so that *DefaultAnnotationHandlerMapping* finds them. Such handler is usually annotated with *RequestMapping* and mapped by *DefaultAnnotationHandlerMapping*. However, it is not compulsory – *AnnotationMethodHandlerAdapter* can execute a handler found by any handler mapping.

I write a simple handler with an annotated method, I call it *CowControllerWithAnnotatedMethod*:

```
package pl.kuku.myspringmvcproject;

import org.springframework.web.bind.annotation.RequestMapping;

public class CowControllerWithAnnotatedMethod {
    @RequestMapping
    public void firstMethod() {
        throw new RuntimeException("muuu");
    }
}
```

Now I configure application context. I create three beans: *CowControllerWithAnnotatedMethod* which will serve as a handler, *SimpleUrlHandlerMapping* which will map requests to this handler and *AnnotationMethodHandlerAdapter* which knows how to handlers with methods annotated with *RequestMapping*:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean id="cow"
class="pl.kuku.myspringmvcproject.CowControllerWithAnnotatedMethod
" />
    <bean
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMap
ping">
        <property name="urlMap">
            <map>
                <entry key="/velez-malaga">
                    <value>cow</value>
                </entry>
            </map>
        </property>
    </bean>
    <bean
class="org.springframework.web.servlet.mvc.annotation.AnnotationMe
thodHandlerAdapter" />
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/velez-malaga* . It works, it throws an exception with a message *muuu*:

```
Request processing failed; nested exception is
java.lang.RuntimeException: muuu
```

Now I'm going to see what will happen when I have in my handler two annotated methods. I add the second method to *CowControllerWithAnnotatedMethod*:

```
package pl.kuku.myspringmvcproject;

import org.springframework.web.bind.annotation.RequestMapping;

public class CowControllerWithAnnotatedMethod {
    @RequestMapping
    public void firstMethod() {
        throw new RuntimeException("muuu");
    }
    @RequestMapping
    public void secondMethod() {
        throw new RuntimeException("mooo");
    }
}
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/velez-malaga* . I get 404 error. I take a look at Tomcat's log, at the file *apache-tomcat-8.0.21/logs/catalina.2015-05-27.log* and there I see this:

```
27-May-2015 10:30:33.986 WARNING [http-nio-8080-exec-156]
org.springframework.web.servlet.mvc.support.DefaultHandlerExceptio
nResolver.handleNoSuchRequestHandlingMethod No matching handler
method found for servlet request: path '/velez-malaga', method
'GET', parameters map[[empty]]
```

But I can have two methods in a handler adapted by *AnnotationMethodHandlerAdapter* – if *RequestMapping* annotations on methods provide URL patterns in such way that the current request matches only one of them.

So, I edit file *mydispatcher-servlet.xml* changing the URL pattern of the *CowControllerWithAnnotatedMethod* (notice that I added an asterisk to that pattern):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean id="cow"
class="pl.kuku.myspringmvcproject.CowControllerWithAnnotatedMethod
" />
    <bean
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMap
ping">
        <property name="urlMap">
            <map>
                <entry key="/velez-malaga/*">
                    <value>cow</value>
                </entry>
            </map>
        </property>
    </bean>
    <bean
class="org.springframework.web.servlet.mvc.annotation.AnnotationMe
thodHandlerAdapter" />
</beans>
```

Then I edit *CowControllerWithAnnotatedMethod* adding URL patterns to methods:

```
package pl.kuku.myspringmvcproject;

import org.springframework.web.bind.annotation.RequestMapping;

public class CowControllerWithAnnotatedMethod {
    @RequestMapping("/velez-malaga/pl")
    public void firstMethod() {
        throw new RuntimeException("muuu");
    }
    @RequestMapping("/velez-malaga/en")
    public void secondMethod() {
        throw new RuntimeException("mooo");
    }
}
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting *http://127.0.0.1:8080/ink/velez-malaga/pl* . It works, it throws an exception with a message *muuu*:

```
Request processing failed; nested exception is
java.lang.RuntimeException: muuu
```

Then I test it by visiting *http://127.0.0.1:8080/ink/velez-malaga/en* . It works, it throws an exception with a message *mooo*:

```
Request processing failed; nested exception is
java.lang.RuntimeException: mooo
```

6.5.3.5.3. *I use the DefaultAnnotationHandlerMapping* handler mapping and the *AnnotationMethodHandlerAdapter* handler adapter together

Now I will use those two things – the *DefaultAnnotationHandlerMapping* handler mapping and the *AnnotationMethodHandlerAdapter* handler adapter – together.

In this experiment I will use JSP templates as views.

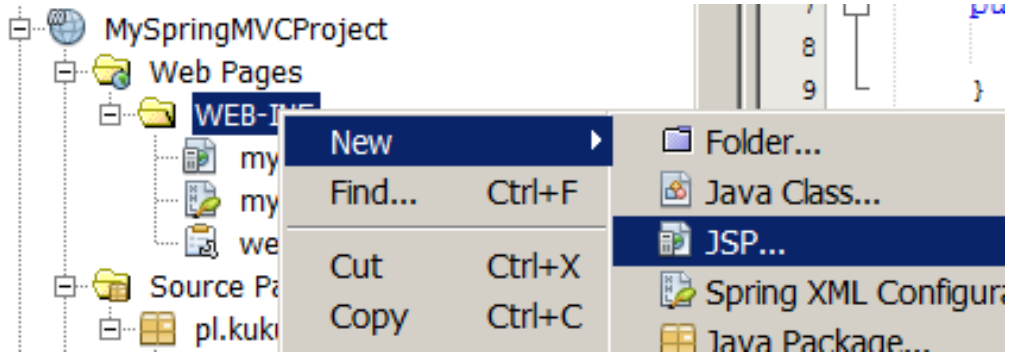
So I will use:

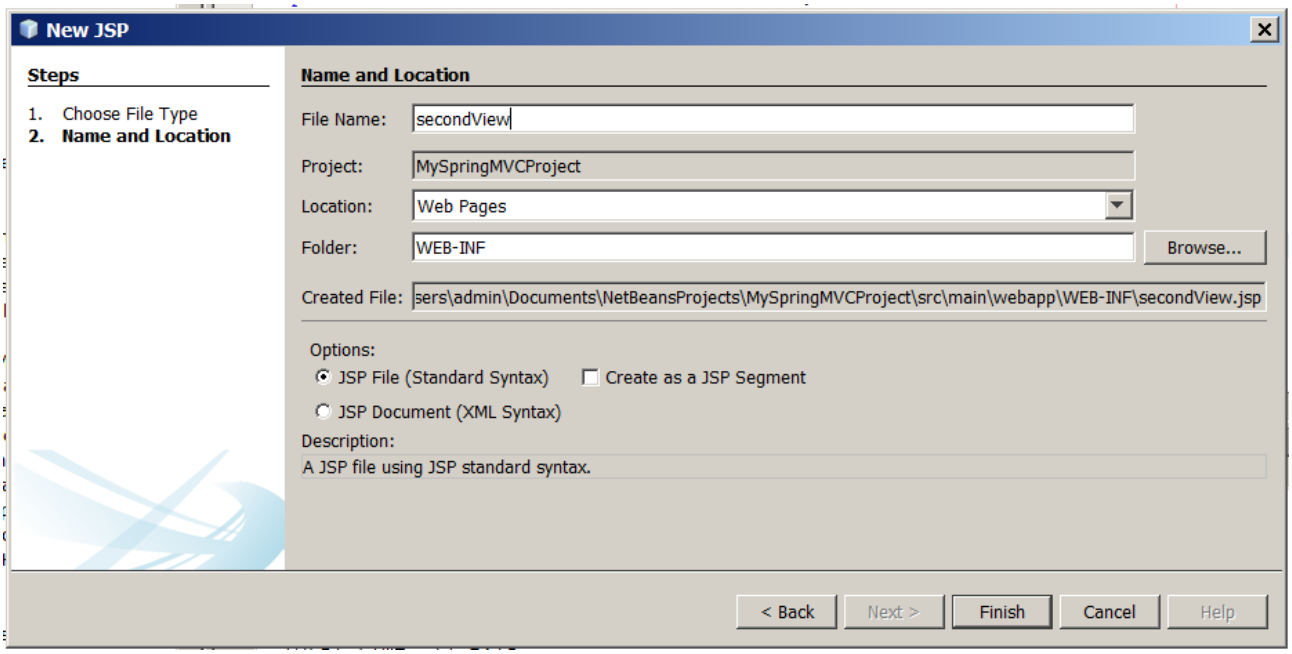
- *DefaultAnnotationHandlerMapping* as a handler mapping,
- handlers written by me, annotated appropriately,
- *AnnotationMethodHandlerAdapter* as a handler adapter,
- *InternalResourceViewResolver* as a view resolver.

6.5.3.5.3.1. I write a JSP template

I create a JSP template displaying some string and some number from the model.

In *Web Pages -> WEB-INF* I create a JSP template called *secondView.jsp*:





In this JSP template I write:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>${text}</h1>
        ${number}
    </body>
</html>
```

In this moment I don't try whether it works. Soon I will write my handlers, I will configure the application context and then I will see if everything works.

6.5.3.5.3.2. I write two annotated handlers with annotated methods

Now I will write two annotated handlers with annotated methods.

I write the first one, I call it *AnnotatedControllerWithAnnotatedMethodsOne*:

```
package pl.kuku.myspringmvcproject;

import java.util.HashMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@RequestMapping("/first")
public class AnnotatedControllerWithAnnotatedMethodsOne {
    @RequestMapping("primera")
    public ModelAndView firstMethod() {
        return new ModelAndView("secondView", new HashMap<String,
Object>() {{
            put("number", 11);
            put("text", "first primera");
        }});
    }
    @RequestMapping("segunda")
    public ModelAndView secondMethod() {
        return new ModelAndView("secondView", new HashMap<String,
Object>() {{
            put("number", 12);
            put("text", "first segunda");
        }});
    }
}
```

Then I write the second one, I call it *AnnotatedControllerWithAnnotatedMethodsTwo* :

```
package pl.kuku.myspringmvcproject;

import java.util.HashMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@RequestMapping("/second")
public class AnnotatedControllerWithAnnotatedMethodsTwo {
    @RequestMapping("primera")
    public ModelAndView firstMethod() {
        return new ModelAndView("secondView", new HashMap<String,
Object>() {{
            put("number", 21);
            put("text", "second primera");
        }});
    }
    @RequestMapping("segunda")
    public ModelAndView secondMethod() {
        return new ModelAndView("secondView", new HashMap<String,
Object>() {{
            put("number", 22);
            put("text", "second segunda");
        }});
    }

}
```

6.5.3.5.3.3. I configure the application context and test if everything works

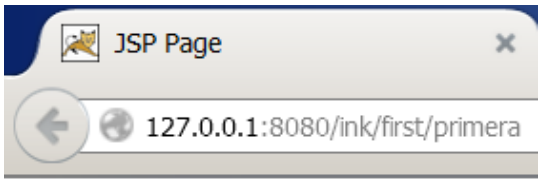
Now I configure the application context (the file *mydispatcher-servlet.xml*):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean
class="org.springframework.web.servlet.mvc.annotation.DefaultAnnot
ationHandlerMapping" />
    <bean
class="pl.kuku.myspringmvcproject.AnnotatedControllerWithAnnotated
MethodsOne" />
    <bean
class="pl.kuku.myspringmvcproject.AnnotatedControllerWithAnnotated
MethodsTwo" />
    <bean
class="org.springframework.web.servlet.mvc.annotation.AnnotationMe
thodHandlerAdapter" />
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewRe
solver">
        <property name="prefix" value="/WEB-INF/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting following URLs:

- *http://127.0.0.1:8080/ink/first/primera*
- *http://127.0.0.1:8080/ink/second/primera*
- *http://127.0.0.1:8080/ink/second/segunda*

It works, I see something like this:



first primera

11

6.5.3.5.3.4. Limitations of *DefaultAnnotationHandlerMapping* and *AnnotationMethodHandlerAdapter*

Spring's documentation (*http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/mvc/annotation/DefaultAnnotationHandlerMapping.html*) says:

Method-level mappings are only allowed to narrow the mapping expressed at the class level (if any). HTTP paths need to uniquely map onto specific handler beans, with any given HTTP path only allowed to be mapped onto one specific handler bean (not spread across multiple handler beans). It is strongly recommended to co-locate related handler methods into the same bean.

I will do some experiments to see if it's true.

6.5.3.5.3.4.1. I do two controllers that are mapped to the same pattern, they only differ by the pattern of their methods

First I will do two controllers that are mapped to the same pattern, they only differ by the pattern of their methods.

I change *AnnotatedControllerWithAnnotatedMethodsOne* :

```
package pl.kuku.myspringmvcproject;

import java.util.HashMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@RequestMapping("/first")
public class AnnotatedControllerWithAnnotatedMethodsOne {
    @RequestMapping("primera")
    public ModelAndView firstMethod() {
        return new ModelAndView("secondView", new HashMap<String,
Object>() {{
            put("number", 11);
            put("text", "first primera");
        }});
    }
}
```

I also change *AnnotatedControllerWithAnnotatedMethodsTwo* :

```
package pl.kuku.myspringmvcproject;

import java.util.HashMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@RequestMapping("/first")
public class AnnotatedControllerWithAnnotatedMethodsTwo {
    @RequestMapping("segunda")
    public ModelAndView secondMethod() {
        return new ModelAndView("secondView", new HashMap<String,
Object>() {{
            put("number", 12);
            put("text", "first segunda");
        }});
    }

}
```

I redeploy the application (by running Maven's *tomcat7:redeploy* goal). I test it by visiting following URLs:

- *http://127.0.0.1:8080/ink/first/primera*
- *http://127.0.0.1:8080/ink/first/segunda*

It works. Hm, it suggests that the documentation is wrong.

6.5.3.5.3.4.2. I do two controllers that are mapped to the same pattern, they only differ that their methods are annotated to react to different HTTP methods

We haven't see it yet, but the *@RequestMapping* annotation can provide not only pattern, but also the HTTP method (get, post, delete etc) that given Java method responds to. Now I will do two controllers that are mapped to the same pattern, they will only differ that their methods are annotated to react to different HTTP methods.

I change *AnnotatedControllerWithAnnotatedMethodsOne* :

```
package pl.kuku.myspringmvcproject;

import java.util.HashMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@RequestMapping("/first")
public class AnnotatedControllerWithAnnotatedMethodsOne {
    @RequestMapping(method = RequestMethod.GET)
    public ModelAndView firstMethod() {
        return new ModelAndView("secondView", new HashMap<String,
Object>() {{
            put("number", 11);
            put("text", "first GET");
        }});
    }
}
```

I change *AnnotatedControllerWithAnnotatedMethodsTwo* :

```
package pl.kuku.myspringmvcproject;

import java.util.HashMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@RequestMapping("/first")
public class AnnotatedControllerWithAnnotatedMethodsTwo {
    @RequestMapping(method = RequestMethod.POST)
    public ModelAndView secondMethod() {
        return new ModelAndView("secondView", new HashMap<String,
Object>() {{
            put("number", 12);
            put("text", "first POST");
        }});
    }
}
```

In Firefox I install the addon called *RESTClient*. With it I send two HTTP requests to *http://127.0.0.1:8080/ink/first* – first using *get* HTTP method, second using *post* HTTP method. In both cases I get 500 error:

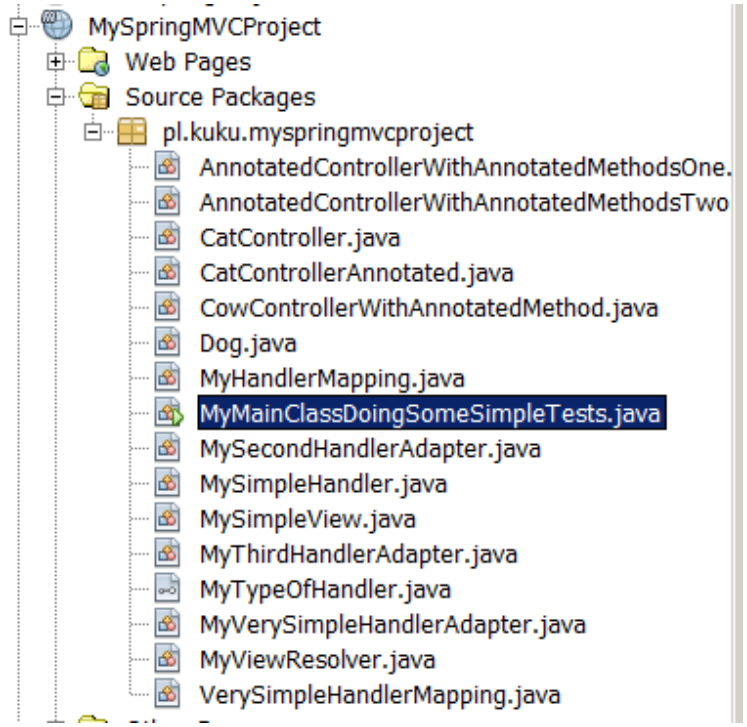
```
java.lang.IllegalStateException: Cannot map handler
'pl.kuku.myspringmvcproject.AnnotatedControllerWithAnnotatedMethod
sTwo#0' to URL path [/first]: There is already handler of type
[class
pl.kuku.myspringmvcproject.AnnotatedControllerWithAnnotatedMethods
One] mapped.
```

It means that documentation is (at least partially) right.

6.5.3.5.3.4.3. I test DefaultAnnotationHandlerMapping without dispatcher servlet

In such situations – like when I want to play with DefaultAnnotationHandlerMapping to see how it behaves when I have two controllers that are mapped to the same pattern, they only differ that their methods are annotated to react to different HTTP methods – it is useful to test it without dispatcher servlet. Now I will do that.

In my project I create a class called *MyMainClassDoingSomeSimpleTests*:




```
package pl.kuku.myspringmvcproject;

public class MyMainClassDoingSomeSimpleTests {
    public static void main(String[] args) {
        System.out.println("hello from
MyMainClassDoingSomeSimpleTests");
    }
}
```

I run it from commandline with command:

```
mvn exec:java -
Dexec.mainClass=pl.kuku.myspringmvcproject.MyMainClassDoingSomeSim
pleTests
```

I see:

```
(...)
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @
MySpringMVCProject ---
hello from MyMainClassDoingSomeSimpleTests
(...)
```

Now I change the *MyMainClassDoingSomeSimpleTests* class to see if I can instantiate the application context and get a bean from it:

```
package pl.kuku.myspringmvcproject;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
import
org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping;

public class MyMainClassDoingSomeSimpleTests {
    public static void main(String[] args) {
        ApplicationContext ctx = new FileSystemXmlApplicationContext("C:\\
Users\\admin\\Documents\\NetBeansProjects\\MySpringMVCProject\\src\\main\\
webapp\\WEB-INF\\mydispatcher-servlet.xml");
        DefaultAnnotationHandlerMapping m =
ctx.getBean(DefaultAnnotationHandlerMapping.class);
        System.out.println("DefaultAnnotationHandlerMapping bean: " + m);
    }
}
```

I run this program with a command:

```
mvn exec:java -
Dexec.mainClass=pl.kuku.myspringmvcproject.MyMainClassDoingSomeSim
pleTests
```

It throws an exception:

```
[ERROR] Failed to execute goal org.codehaus.mojo:exec-maven-
plugin:1.4.0:java (default-cli) on project MySpringMVCProject: An
exception occured while executing the Java class. null:
InvocationTargetException: Error creating bean with name
'org.springframework.web.servlet.mvc.annotation.DefaultAnnotationH
andlerMapping#0' defined in file [C:\Users\admin\Documents\
NetBeansProjects\MySpringMVCProject\src\main\webapp\WEB-INF\
mydispatcher-servlet.xml]: Initialization of bean failed; nested
exception is java.lang.IllegalStateException: Cannot map handler
'pl.kuku.myspringmvcproject.AnnotatedControllerWithAnnotatedMethod
sTwo#0' to URL path [/first]: There is already handler of type
[class
pl.kuku.myspringmvcproject.AnnotatedControllerWithAnnotatedMethods
One] mapped. -> [Help 1]
```

I have this error because I still have two controllers that have the same pattern. I comment out the *@RequestMapping* annotation on *AnnotatedControllerWithAnnotatedMethodsTwo* class:

```
package pl.kuku.myspringmvcproject;

import java.util.HashMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

//@RequestMapping("/first")
public class AnnotatedControllerWithAnnotatedMethodsTwo {
    @RequestMapping(method = RequestMethod.POST)
    public ModelAndView secondMethod() {
        return new ModelAndView("secondView", new HashMap<String,
Object>() {{
            put("number", 12);
            put("text", "first POST");
        }});
    }
}
```

Now I run my program again. This time it runs and prints a lot of useful information:

```
(...)
[INFO] --- exec-maven-plugin:1.4.0:java (default-cli) @
MySpringMVCProject ---
maj 30, 2015 4:13:10 PM
org.springframework.context.support.AbstractApplicationContext
prepareRefresh
INFO: Refreshing
org.springframework.context.support.FileSystemXmlApplicationContex
t@29d89: startup date [Sat May 30 16:13:10 CEST 2015]; root of
context hierarchy
maj 30, 2015 4:13:10 PM
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Users\admin\
Documents\NetBeansProjects\MySpringMVCProject\src\main\webapp\WEB-
INF\mydispatcher-servlet.xml]
maj 30, 2015 4:13:10 PM
org.springframework.web.servlet.handler.AbstractUrlHandlerMapping
registerHandler
INFO: Mapped URL path [/first] onto handler
'pl.kuku.myspringmvcproject.AnnotatedControllerWithAnnotatedMethod
sOne#0'
maj 30, 2015 4:13:10 PM
org.springframework.web.servlet.handler.AbstractUrlHandlerMapping
registerHandler
INFO: Mapped URL path [/first.*] onto handler
'pl.kuku.myspringmvcproject.AnnotatedControllerWithAnnotatedMethod
sOne#0'
maj 30, 2015 4:13:10 PM
org.springframework.web.servlet.handler.AbstractUrlHandlerMapping
registerHandler
INFO: Mapped URL path [/first/] onto handler
'pl.kuku.myspringmvcproject.AnnotatedControllerWithAnnotatedMethod
sOne#0'
DefaultAnnotationHandlerMapping bean:
org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHa
ndlerMapping@133ea67
(...)
```

Now, when I can get the *DefaultAnnotationHandlerMapping* bean from the application context, I will try to see how it behaves when I call its *getHandler* method. However, when calling this method I will have to provide it the request object – the object of a class implementing the *HttpServletRequest* interface. It is tedious to implement this interface, because it requires a lot of methods. Spring has a lot of mockup classes for such situations, for instance it has a *MockHttpServletRequest* class.

So, I add do my *pom.xml* a dependency with that class:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>pl.kuku</groupId>
    <artifactId>MySpringMVCProject</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>
    <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.7</maven.compiler.source>
        <maven.compiler.target>1.7</maven.compiler.target>
    </properties>
    <build>
        <finalName>${project.artifactId}</finalName>
        <plugins>
            <plugin>
                <groupId>org.apache.tomcat.maven</groupId>
                <artifactId>tomcat7-maven-plugin</artifactId>
                <version>2.2</version>
                <configuration>
                    <url>http://localhost:8080/manager/text</url>
                    <server>TomcatServer</server>
                    <path>/ink</path>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>4.0.1.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>4.0.1.RELEASE</version>
        </dependency>
```

```
        <dependency>
            <groupId>org.jboss.spec.javax.servlet</groupId>
            <artifactId>jboss-servlet-api_3.0_spec</artifactId>
            <version>1.0.2.Final</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

Now I add in *MyMainClassDoingSomeSimpleTests* a fragment which calls *getHandler* on the *DefaultAnnotationHandlerMapping* bean and prints out the result:

```
package pl.kuku.myspringmvcproject;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplicationContex
t;
import org.springframework.mock.web.MockHttpServletRequest;
import org.springframework.web.servlet.HandlerExecutionChain;
import
org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHa
ndlerMapping;

public class MyMainClassDoingSomeSimpleTests {
    public static void main(String[] args) throws Exception {
        ApplicationContext ctx = new
FileSystemXmlApplicationContext("C:\\Users\\admin\\Documents\\
NetBeansProjects\\MySpringMVCProject\\src\\main\\webapp\\WEB-INF\\
mydispatcher-servlet.xml");
        DefaultAnnotationHandlerMapping m =
ctx.getBean(DefaultAnnotationHandlerMapping.class);
        System.out.println("DefaultAnnotationHandlerMapping bean:
" + m);
        MockHttpServletRequest r = new
MockHttpServletRequest("GET", "/first");
        HandlerExecutionChain h = m.getHandler(r);
        System.out.println("handler=" + h);
    }
}
```

Again I run my program. It works:

```
(...)
handler=HandlerExecutionChain with handler
[pl.kuku.myspringmvcproject.AnnotatedControllerWithAnnotatedMethod
sOne@333027] and 1 interceptor
```

This technique – running parts of my application without the dispatcher servlet – can be very useful during debugging and learning.

6.5.3.5.3.4.4. I take a look at the mapping that DefaultAnnotationHandlerMapping uses to map requests to handlers

Now I will take a look at the mapping that DefaultAnnotationHandlerMapping uses to map requests to handlers.

First I download sources of all the dependencies my project uses. I do it by running this maven goal:

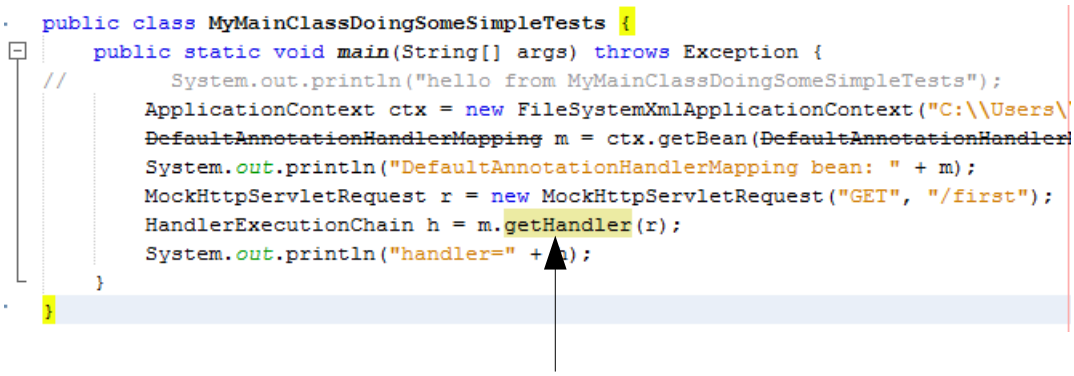
```
dependency:sources
```

The output of this command is:

```
------------------------------------------------------------------------
Building MySpringMVCProject 1.0-SNAPSHOT
------------------------------------------------------------------------

--- maven-dependency-plugin:2.1:sources (default-cli) @
MySpringMVCProject ---

The following files have been resolved:
   none

The following files were skipped:
   aopalliance:aopalliance:java-source:sources:1.0
   commons-logging:commons-logging:java-source:sources:1.1.1
   javax.servlet:jsp-api:java-source:sources:2.0
   javax.servlet:servlet-api:java-source:sources:2.4
   org.jboss.spec.javax.servlet:jboss-servlet-api_3.0_spec:java-
source:sources:1.0.2.Final
   org.springframework:spring-aop:java-
source:sources:4.0.1.RELEASE
   org.springframework:spring-beans:java-
source:sources:4.0.1.RELEASE
   org.springframework:spring-context:java-
source:sources:4.0.1.RELEASE
   org.springframework:spring-core:java-
source:sources:4.0.1.RELEASE
   org.springframework:spring-expression:java-
source:sources:4.0.1.RELEASE
   org.springframework:spring-test:java-
source:sources:4.0.1.RELEASE
   org.springframework:spring-web:java-
source:sources:4.0.1.RELEASE
   org.springframework:spring-webmvc:java-
source:sources:4.0.1.RELEASE
```

It means that somebody who packed the Spring Framework for Maven did not package its sources. So I have to download it from somewhere else[7]. The sources of Spring Framework are on Github. I can read the sources of *AbstractHandlerMapping* class and other classes online at *https://github.com/spring-projects/spring-framework/blob/master/spring-webmvc/src/main/java/org/springframework/web/servlet/handler/AbstractHandlerMapping.java* . I can also download the whole sources of Spring Framework from *https://github.com/spring-projects/spring-framework* (the zip is at *https://github.com/spring-projects/spring-framework/archive/master.zip*). So I download that *master.zip* and I attach it to *spring-webmvc-4.0.1.RELEASE.jar*. Maybe it is possible to do it more simply, but I had to unzip that *master.zip*, inside it find directory find the *spring-framework-master/spring-webmvc/src/main/java*, zip the contents of this directory (nb: I have to zip the contents of this directory, not the directory itself – I create a jar), rename this zip to *something.jar*, right-click on *spring-webmvc-4.0.1.RELEASE.jar* (in Netbeans), choose *add local sources* and point netbeans to the jar I just created.

Now in Netbeans I control click on the method *getHandler*:



and I can read the sources of this method and its surroundings.

So I read the sources of this method and its surroundings and I find that *DefaultAnnotationHandlerMapping* extends *org.springframework.web.servlet.handler.AbstractUrlHandlerMapping* and that *AbstractUrlHandlerMapping* has a field called *handlerMap*. This *handlerMap* is of type *Map<String, Object>*. Its keys are URL patterns and its values are handlers (or handlers' names).

I can take a look at how this *handlerMap* looks like when I have two annotated controllers that are mapped to the same pattern, they only differ that their methods are annotated to react to different patterns.

7 This was (maybe) true in 2015, when I first wrote this document. Now, in 2019, sources of Spring Web MVC can be downloaded with maven. It can be easily done in maven by right clicking on jar and choosing *download sources*.

I edit the class *AnnotatedControllerWithAnnotatedMethodsOne* so it looks like that:

```
package pl.kuku.myspringmvcproject;

import java.util.HashMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@RequestMapping("/first")
public class AnnotatedControllerWithAnnotatedMethodsOne {
    @RequestMapping("primera")
    public ModelAndView firstMethod() {
        return new ModelAndView("secondView", new HashMap<String,
Object>() {{
            put("number", 11);
            put("text", "first GET");
        }});
    }
}
```

I also edit the class *AnnotatedControllerWithAnnotatedMethodsTwo* so it looks like that:

```
package pl.kuku.myspringmvcproject;

import java.util.HashMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@RequestMapping("/first")
public class AnnotatedControllerWithAnnotatedMethodsTwo {
    @RequestMapping("segunda")
    public ModelAndView secondMethod() {
        return new ModelAndView("secondView", new HashMap<String,
Object>() {{
            put("number", 12);
            put("text", "first POST");
        }});
    }
}
```

I also make sure that *mydispatcher-servlet.xml* still looks like that:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean
class="org.springframework.web.servlet.mvc.annotation.DefaultAnnot
ationHandlerMapping" />
    <bean
class="pl.kuku.myspringmvcproject.AnnotatedControllerWithAnnotated
MethodsOne" />
    <bean
class="pl.kuku.myspringmvcproject.AnnotatedControllerWithAnnotated
MethodsTwo" />
    <bean
class="org.springframework.web.servlet.mvc.annotation.AnnotationMe
thodHandlerAdapter" />
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewRe
solver">
        <property name="prefix" value="/WEB-INF/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

Now in *MyMainClassDoingSomeSimpleTests* I add a few lines which print out the field *handlerMap* from an object of class *DefaultAnnotationHandlerMapping*. I must do it using introspection because it is not public:

```
package pl.kuku.myspringmvcproject;

import java.lang.reflect.Field;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplicationContex
t;
import org.springframework.mock.web.MockHttpServletRequest;
import org.springframework.web.servlet.HandlerExecutionChain;
import
org.springframework.web.servlet.handler.AbstractUrlHandlerMapping;
import
org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHa
ndlerMapping;

public class MyMainClassDoingSomeSimpleTests {
    public static void main(String[] args) throws Exception {
        ApplicationContext ctx = new
FileSystemXmlApplicationContext("C:\\Users\\Excel\\Desktop\\
psobolewski\\java_ee\\spring_notatki\\MySpringMVCProject\\src\\
main\\webapp\\WEB-INF\\mydispatcher-servlet.xml");
        DefaultAnnotationHandlerMapping m =
ctx.getBean(DefaultAnnotationHandlerMapping.class);
        System.out.println("DefaultAnnotationHandlerMapping bean:
" + m);
        MockHttpServletRequest r = new
MockHttpServletRequest("GET", "/first");
        HandlerExecutionChain h = m.getHandler(r);
        System.out.println("handler=" + h);
        Field handlerMapField =
AbstractUrlHandlerMapping.class.getDeclaredField("handlerMap");
        handlerMapField.setAccessible(true);
        Object handlerMap = handlerMapField.get(m);
        System.out.println("handlerMap = " + handlerMap);
    }
}
```

I run this program by running maven's goal *exec:java -Dexec.mainClass=pl.kuku.myspringmvcproject.MyMainClassDoingSomeSimpleTests* . It prints out:

```
handlerMap =
{/first/primera=pl.kuku.myspringmvcproject.AnnotatedControllerWith
AnnotatedMethodsOne@41ef27bf,
/first/primera.*=pl.kuku.myspringmvcproject.AnnotatedControllerWit
hAnnotatedMethodsOne@41ef27bf,
/first/primera/=pl.kuku.myspringmvcproject.AnnotatedControllerWith
```

```
AnnotatedMethodsOne@41ef27bf,
/first/segunda=pl.kuku.myspringmvcproject.AnnotatedControllerWithA
nnotatedMethodsTwo@682bd2de,
/first/segunda.*=pl.kuku.myspringmvcproject.AnnotatedControllerWit
hAnnotatedMethodsTwo@682bd2de,
/first/segunda/=pl.kuku.myspringmvcproject.AnnotatedControllerWith
AnnotatedMethodsTwo@682bd2de}
```

Everything is clear now. In the mapping of *DefaultAnnotationHandlerMapping* keys are URL patterns. So we just can't have two controllers reacting to the same URL pattern.

6.5.3.6. I use the *RequestMappingHandlerMapping* handler mapping and the *RequestMappingHandlerAdapter* handler adapter

As of 2019 (and Spring Framework version 5) *DefaultAnnotationHandlerMapping* and *AnnotationMethodHandlerAdapter* are not enabled by default by the properties file of the dispatcher servlet. They are replaced by *RequestMappingHandlerMapping* and *RequestMappingAdapter*.

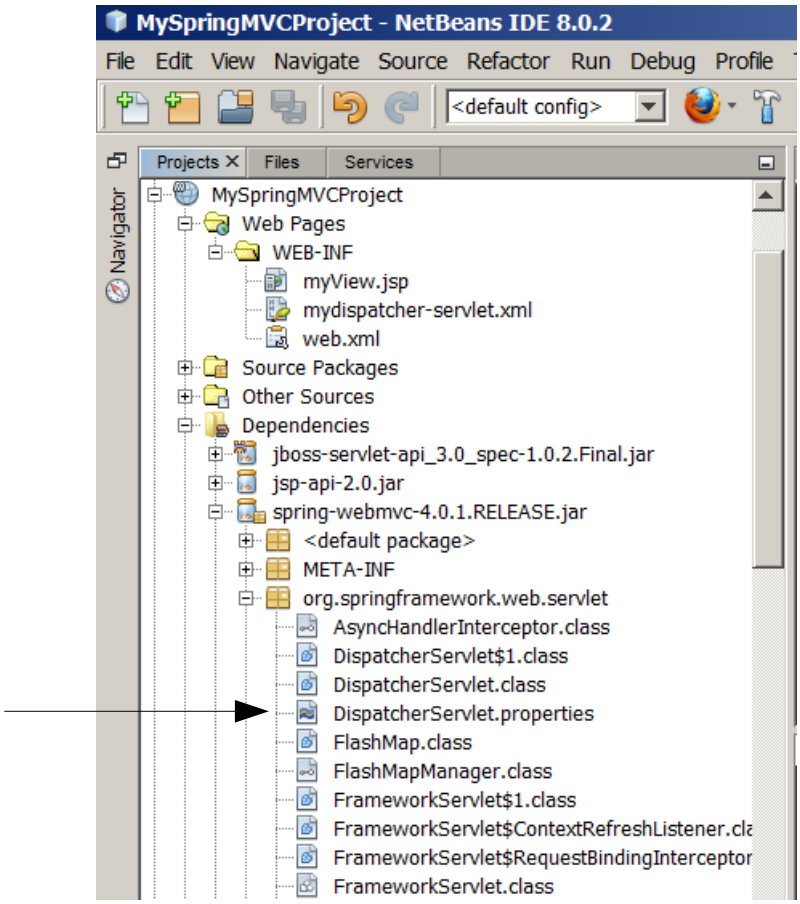
RequestMappingHandlerMapping is a handler mapping which finds all beans annotated (at class level) with *@Controller* or *@RequestMapping* annotation, finds in them all methods annotated with *@RequestMapping* and then for each of these methods combines the filter (path, HTTP method etc) from class level *@RequestMapping* with the filter from method level *@RequestMapping* and for a request matching this filter returns a handler (of type *HandlerMethod*) which can call this method. There also exists a handler adapter – called *RequestMappingHandlerAdapter* – which can execute handlers of type *HandlerMethod*.

FIXME: here I should have an example which uses *RequestMappingHandlerMapping* handler mapping and the *RequestMappingHandlerAdapter* handler adapter.

FIXME: I should mention @RestController, @RequestBody and @ResponseBody here.

6.6. Dispatcher servlet has a property file with a default configuration

Dispatcher servlet has a property file with a default configuration. We can see this file here:



This properties file decides which handler mappings will be used when context does not contain any handler mapping beans, which handler adapters will be used when context does not contain any handler adapter beans etc.

6.7. todo

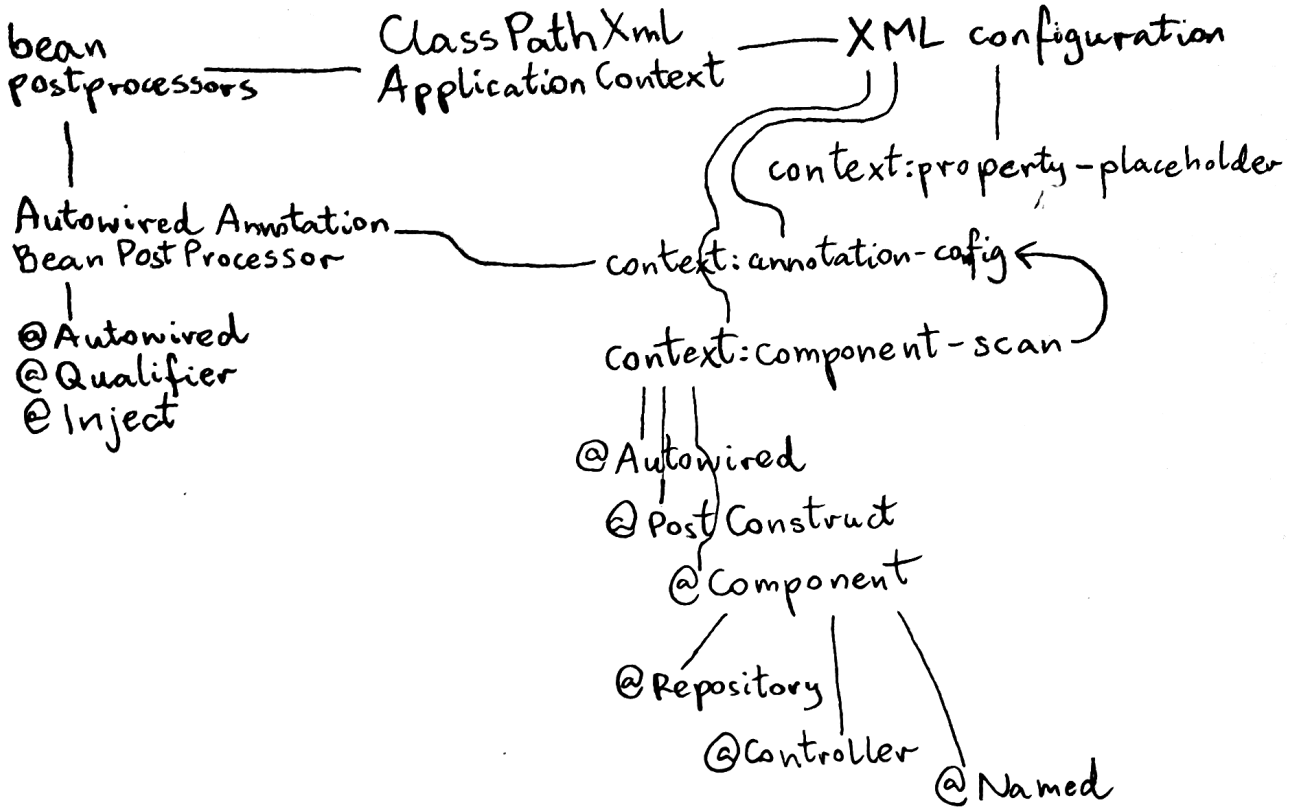
FIXME: write about beans' scopes: https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans-factory-scopes

FIXME: write that in web mvc Spring applications we can configure Spring with annotations instead of XML: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html#beans-java-instantiating-container-web

7. Summary

Here we have a short summary of how some parts of Spring Framework work.

7.1. Core

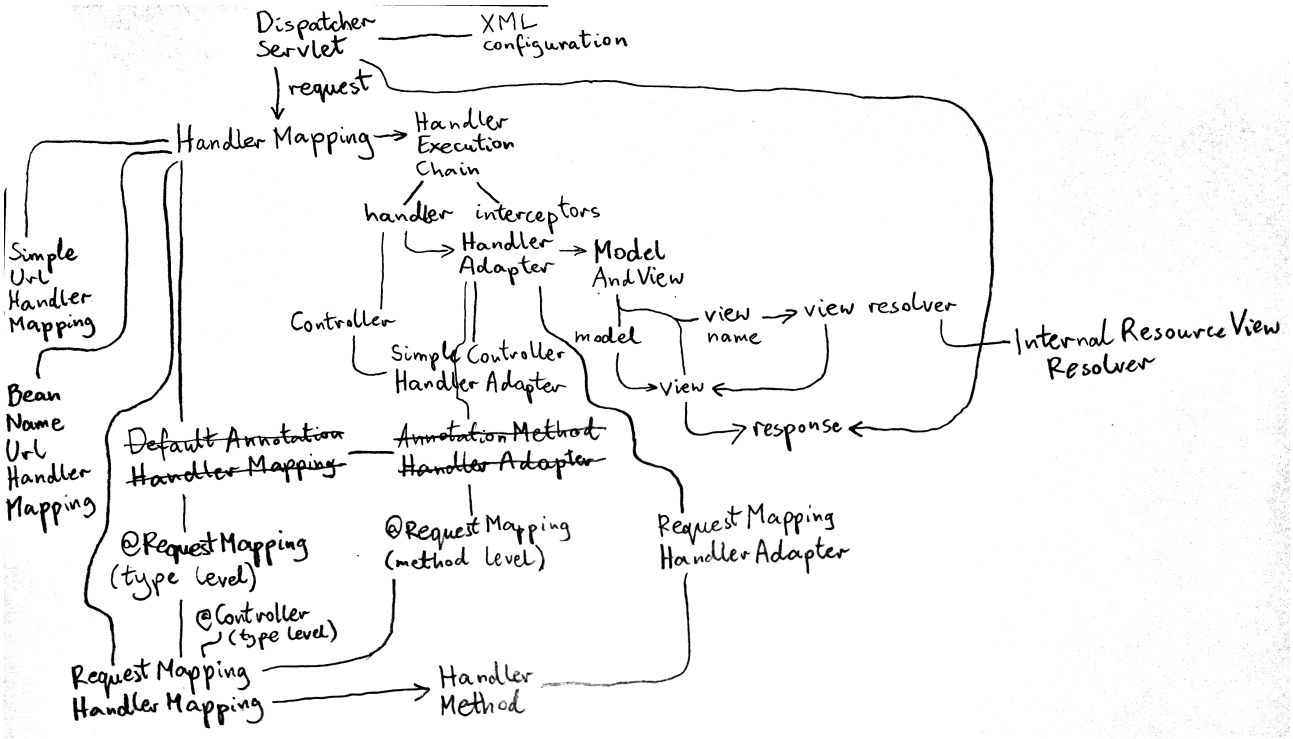


ClassPathXmlApplicationContext (a.k.a. context) is a container which creates objects, configures them (by setting their properties), stores them and let the programmer get them. This container has XML configuration which says which objects should be created and how they should should be connected. Configuration can contain *context:property-placeholder* tag which lets us include and use property files in the configuration.

The context can have bean postprocessors configured: such postprocessor is a class which is called whenever context creates a bean, postprocessor can then alter the bean. An interesting postprocessor is *AutowiredAnnotationBeanPostProcessor* which lets inject dependencies into beans with annotations: *@Autowired*, *@Qualifier*, *@Inject*. This postprocessor can be enabled with *context:annotation-config* tag. There is also another interesting configuration tag: *context:component-scan*. This tag enables *AutowiredAnnotationBeanPostProcessor* and also scans all Java packages, finds classes annotated with *@Component* (or: *@Repository*, *@Controller* or *@Named*) and adds their

instances to the context. It also supports *@PostConstruct* annotation: if a bean contains a method annotated with this annotation, this method is executed after the bean has been created and its dependencies has been injected.

7.2. Web MVC



DispatcherServlet is a web servlet which receives HTTP requests, executes some handler (which consumes request and produces response's contents) and returns HTTP response.

When *DispatcherServlet* receives request, it passes it to handler mappings. One of handler mappings decide that it can handle this request: it produces handler execution chain, which consists of any number of interceptors and one handler. Then dispatcher servlet passes the handler to handler adapters. One of handler adapters decides that it can execute this handler. It executes the handler, gets the result returned by the handler, puts the result into a model and returns an object called ModelAndView. ModelAndView contains two things:

- model
- either view (which is an object which can turn a model into HTTP response) or view name

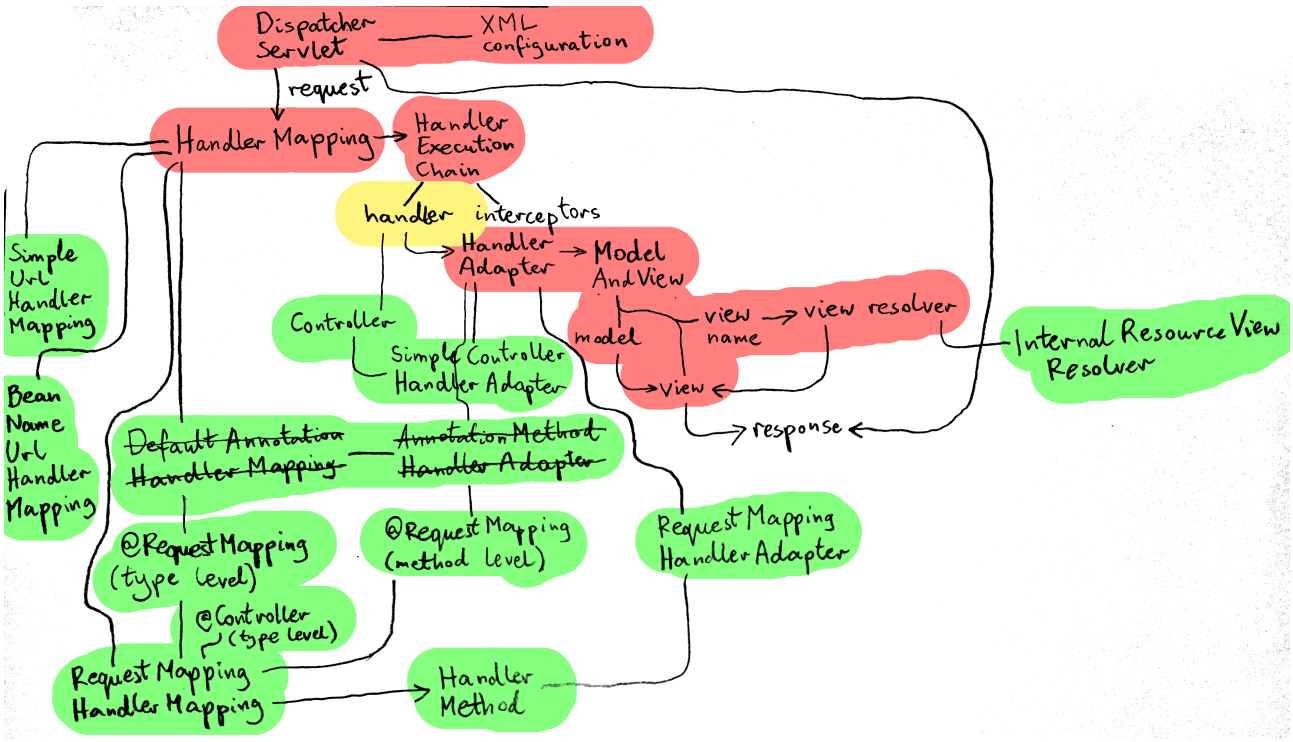
If ModelAndView returned by handler adapter contains view name, dispatcher servlet passes this name to view resolvers. One of view resolvers decide that it can handle this view name and returns a view.

Dispatcher servlet passes the model to the view. It also passes response object to the view. View converts data from model to the contents of HTTP response and puts it into response object which it got from the dispatcher servlet. Dispatcher servlet uses the response object to produce HTTP response and send it to HTTP client.

When we make web applications with Spring Web MVC, we deal with three types of stuff (classes, interfaces, annotations etc):

- stuff created by authors of Spring – it is a part of Spring Web MVC framework, you must use it when you use Spring
- stuff which you yourself must create – it contains the business logic of your web application, nobody but you can make it
- stuff which in theory you could write yourself, but in practice it makes more sense is to use something provided by Spring authors – it connects your business logic with Spring MVC

On below picture stuff which you must create is yellow, stuff which you must not create is red and stuff which you could create but it is better to use what Spring authors have provided is green:



7.2.1. Stuff which you don't want to write yourself

Stuff which you don't want to write yourself (the green stuff on the picture) is used (by Spring Web MVC framework) to connect the framework (red) with our business logic (yellow). Authors of Spring have decided that it is a good idea to allow programmers to write business logic in different ways, they created a lot of green stuff. For instance, in your web application you only need one HandlerAdapter (to connect the framework with your handler), but you can choose from several different handler adapters – with each of them you write your handlers in a different way.

Controller is a simple interface for handlers: it contains a method which must produce handler's response. Attention: it has nothing to do with *@Controller* annotation. *SimpleControllerHandlerAdapter* is a handler adapter which can

SimpleUrlHandlerMapping is a handler mapping which has a map which maps URLs to handlers. You configure it by setting this map in the XML configuration of application context, by setting a property of *SimpleUrlHandlerMapping* bean.

BeanNameUrlHandlerMapping is a handler mapping which treats bean names as patterns, it compares the URL with the pattern and returns the bean if the pattern matches.

~~*DefaultAnnotationHandlerMapping*~~ and ~~*AnnotationMethodHandlerAdapter*~~ is a deprecated pair of handler mapping and handler adapter. ~~*DefaultAnnotationHandlerMapping*~~ chooses a class which will be used as a handler based on the pattern provided in *@RequestMappping* annotation on this class. ~~*AnnotationMethodHandlerAdapter*~~ calls one of methods of this class – which method should be used is chosen based on patterns provided in *@RequestMapping* annotation on methods.

This deprecated pair is replaced with *RequestMappingHandlerMapping* and *RequestMappingHandlerAdapter*. *RequestMappingHandlerMapping* iterates through all beans annotated with *@Controller* or *@RequestMapping*, for each bean it iterates through all its methods annotated with *@RequestMapping*, it uses patterns provided in these *@RequestMapping* annotations to choose the method which will handle the request and produces a handler (of type *HandlerMethod*) which call this method. Then *RequestMappingHandlerAdapter* handler adapter can execute this *HandlerMethod* handler.

InternalResourceViewResolver is a view resolver which produces a view which redirects a request to a resource (for instance, to JSP template).