№2/2 Spring MVC Configuration (без Spring Boot)

Конфигурация Spring

Настроить проект Spring MVC, можно с конфигурацией на основе Java, так и с конфигурацией XML. Предположим, Spring Boot не используется

1) Зависимости Maven для проекта Spring MVC

```
<properties>
    <spring-framework-version>5.2.7.RELEASE</spring-framework-version>
    <java-version>1.8</java-version>
    <org.slf4j-version>1.7.30</org.slf4j-version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring-framework-version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring-framework-version}</version>
    </dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring-framework-version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>${spring-framework-version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring-framework-version}</version>
</dependency>
```

и т.д.

Зависимость Spring-Web содержит общие веб-утилиты для сред Servlet и Portlet, а Spring-Webmvc обеспечивает поддержку MVC для сред Servlet.

Поскольку у Spring-webmvc есть Spring-Web как зависимость, явное определение Spring-Web не требуется при использовании Spring-Webmvc.

2) Теперь рассмотрим как сконфигурировать проект используя XML файл. Начнем с servlet-context.xml именно этот файл будет конфигурировать проект (название может быть другое), в основном тут мы описываем все наши beans.

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

   <!-- DispatcherServlet Context: defines this servlet's request-processing
      infrastructure -->

   <!-- Enables the Spring MVC @Controller programming model -->
   <mvc:annotation-driven/>

   <!-- Handles HTTP GET requests for /resources/** by efficiently serving
      up static resources in the ${webappRoot}/resources directory -->
   <mvc:resources mapping="/resources/**" location="/resources/" />
   <context:component-scan base-package="by.pnv" />

   <bean id="propertyConfigurer"
         class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
      <property name="location" value="/WEB-INF/messages.properties" />
   </bean>


   <!-- Resolves views selected for rendering by @Controllers to .jsp resources
      in the /WEB-INF/views directory -->
   <bean
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <property name="prefix" value="/WEB-INF/views/" />
      <property name="suffix" value=".jsp" />
   </bean>


   <mvc:interceptors>
      <mvc:interceptor>
         <mvc:mapping path="/*" />
         <bean class="by.pnv.interceptors.CheckUserInterceptor" />
      </mvc:interceptor>
   </mvc:interceptors>
```

В данной конфигурации мы указываем, где искать все наши контроллеры, сервисы и другие компоненты с помощью тега: context:component-scan, а также инициализируем InternalResourceViewResolver, который отвечайте за показ View в нашем случае это jsp страницы.

Если мы хотим использовать чисто XML-конфигурацию, нам также нужно добавить файл *web.xml* для начальной загрузки приложения.

```
 <listener>
       <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
```

Это ServletContextListener, он срабатывает на инициализацию Servlet Context'a для веб приложения.

Далее надо указать настройку, которая расскажет Spring, где лежит его особый xml конфиг с настройками:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
    <!--        <param-value>/WEB-INF/applicationContext.xml</param-value>-->
</context-param>
```

Будет использована Spring'ом при инициализации Application Context.

```
<mvc:annotation-driven/>
```

Для получения подробной информации см.
https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc

```
<context:component-scan base-package="by.pnv" />
```

Включает функцию автоматического сканирования в Spring. Базовый пакет указывает, где хранятся ваши компоненты, Spring просканирует эту папку и найдет bean-компонент (помеченный @Component) и зарегистрирует его в контейнере Spring.

```
<mvc:resources mapping="/resources/**" location="/resources/" />
```

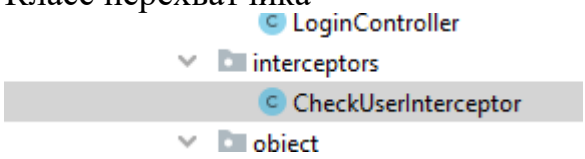
Вы можете использовать элемент mvc: resources, чтобы указать расположение ресурсов с определенным общедоступным шаблоном URL.

Будет обслуживать все запросы на ресурсы, поступающие с шаблоном общедоступного URL-адреса, например «/ resources / **», путем поиска в каталоге «/ resources /» в корневой папке нашего приложения.

Рассмотрим перехватчики

```
<mvc:interceptors>
      <mvc:interceptor>
         <mvc:mapping path="/*" />
         <bean class="by.pnv.interceptors.CheckUserInterceptor" />
      </mvc:interceptor>
 </mvc:interceptors>
```

Класс перехватчика




```
import by.pnv.object.User;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;


import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class CheckUserInterceptor extends HandlerInterceptorAdapter { //
implements HandlerInterceptor{

   @Override
   public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
      if (request.getRequestURI().contains("check-user")) {

         User user = (User) modelAndView.getModel().get("user");
         if (user == null || !user.isAdmin()) {
            response.sendRedirect(request.getContextPath() + "/failed");
         }

      }
   }

}
```

может выполнять определенный код до вызова методов контроллера, после него, а так же по завершению запроса.

web.xml

Одним из основных понятий Spring MVC является DispatcherServlet.

DispatcherServlet является точкой входа каждого приложения Spring MVC. Его цель - перехватывать HTTP-запросы и отправлять их нужному компоненту, который будет знать, как с ним работать.

До Spring 3.1 единственным способом настройки DispatcherServlet был файл WEB-INF / web.xml. В этом случае требуется два шага.

Первым шагом является объявление сервлета:

```
    <servlet>

        <servlet-name>appServlet</servlet-name>
        <!--                    <servlet-name>dispatcher</servlet-name>-->
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/servlet-context.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
```

Если вам нужно объявить более одного сервлета, вы можете определить, в каком порядке они будут инициализированы. Сервлеты, отмеченные более низкими целыми числами, загружаются перед сервлетами, отмеченными более высокими целыми числами.

Второй шаг - объявление отображения сервлета:

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

При отображении сервлета мы привязываем его по имени к шаблону URL, который указывает, какие HTTP-запросы будут обрабатываться им.

Интерфейс BeanPostProcessor

На схеме изображены основные этапы поднятия ApplicationContext.

Парсирование конфигурации и создание BeanDefinition

После выхода четвертой версии спринга, у нас появилось несколько способов конфигурирования контекста:

1. Xml конфигурация — ClassPathXmlApplicationContext(“context.xml”)
2. Конфигурация через аннотации с указанием пакета для сканирования — AnnotationConfigApplicationContext(“package.name”)
3. Конфигурация через аннотации с указанием класса (или массива классов) помеченного аннотацией @Configuration - AnnotationConfigApplicationContext(JavaConfig.class). Этот способ конфигурации называется — JavaConfig.

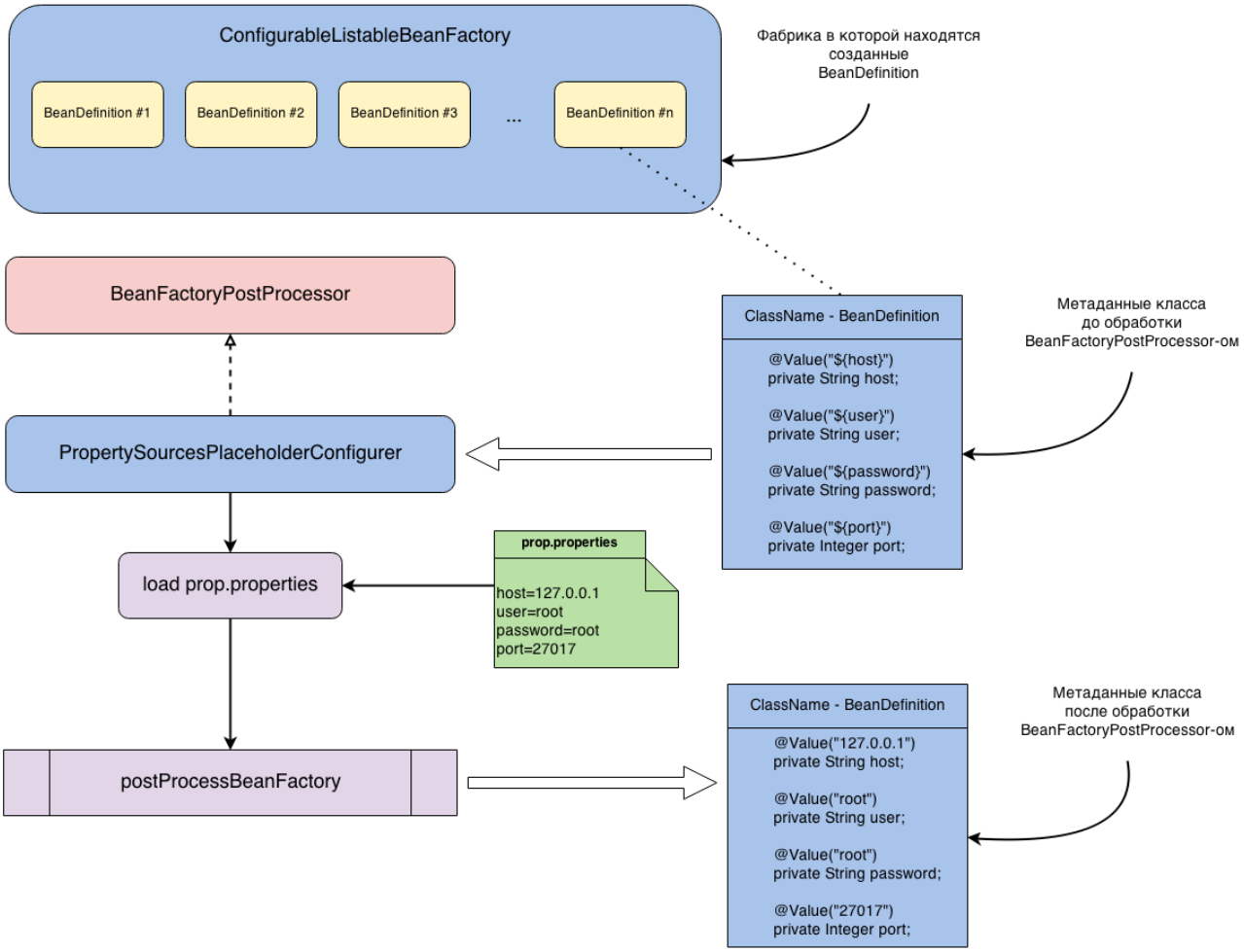
Цель первого этапа — это создание всех BeanDefinition. BeanDefinition — это специальный интерфейс, через который можно получить доступ к метаданным будущего бина. В зависимости от того, какая у вас конфигурация, будет использоваться тот или иной механизм парсирования конфигурации.

Настройка созданных BeanDefinition

После первого этапа у нас имеется Map, в котором хранятся *BeanDefinition*. Архитектура спринга построена таким образом, что у нас есть возможность повлиять на то, какими будут наши бины еще до их фактического создания, иначе говоря мы имеем доступ к метаданным класса. Для этого существует специальный интерфейс *BeanFactoryPostProcessor*, реализовав который, мы получаем доступ к созданным *BeanDefinition* и можем их изменять. В этом интерфейсе всего один метод.

```
public interface BeanFactoryPostProcessor {

void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws

BeansException;

}
```

Обычно, настройки подключения к базе данных выносятся в отдельный property файл, потом при помощи *PropertySourcesPlaceholderConfigurer* они загружаются и делается inject этих значений в нужное поле. Так как inject делается по ключу, то до создания экземпляра бина нужно заменить этот ключ на само значение из property файла. Эта замена происходит в классе, который реализует интерфейс *BeanFactoryPostProcessor*. Название этого класса — *PropertySourcesPlaceholderConfigurer*. Весь этот процесс можно увидеть на рисунке ниже.



Для того что бы *PropertySourcesPlaceholderConfigurer* был добавлен в цикл настройки созданных *BeanDefinition*, нужно сделать одно из следующих действий.

Для XML конфигурации.

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

Для JavaConfig.

```
@Configuration
@PropertySource("classpath:property.properties")
public class DevConfig {
	@Bean
	public static PropertySourcesPlaceholderConfigurer configurer() {
	    return new PropertySourcesPlaceholderConfigurer();
	}
}
```

Создание кастомных FactoryBean

FactoryBean — это generic интерфейс, которому можно делегировать процесс создания бинов типа . В те времена, когда конфигурация была исключительно в xml, разработчикам был необходим механизм с помощью которого они бы могли управлять процессом создания бинов.

```
<?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.sp
ringframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schem
a/context http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="redColor" scope="prototype" class="java.awt.Color">
        <constructor-arg name="r" value="255" />
        <constructor-arg name="g" value="0" />
        <constructor-arg name="b" value="0" />
    </bean>

</beans>
```

```
<?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.sp
ringframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schem
a/context http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="redColor" scope="prototype" class="java.awt.Color">
        <constructor-arg name="r" value="255" />
        <constructor-arg name="g" value="0" />
        <constructor-arg name="b" value="0" />
    </bean>

    <bean id="green" scope="prototype" class="java.awt.Color">
        <constructor-arg name="r" value="0" />
        <constructor-arg name="g" value="255" />
        <constructor-arg name="b" value="0" />
    </bean>

</beans>
```

Если нужен каждый раз случайный цвет на помощь интерфейс *FactoryBean*. Создадим фабрику которая будет отвечать за создание всех бинов типа — *Color*.

```
package com.malahov.factorybean;

import org.springframework.beans.factory.FactoryBean;
import org.springframework.stereotype.Component;
```

```
import java.awt.*;
import java.util.Random;


public class ColorFactory implements FactoryBean<Color> {
    @Override
    public Color getObject() throws Exception {
        Random random = new Random();
        Color color = new Color(random.nextInt(255), random.nextInt(255), random.nex
tInt(255));
        return color;
    }

    @Override
    public Class<?> getObjectType() {
        return Color.class;
    }

    @Override
    public boolean isSingleton() {
        return false;
    }
}
```

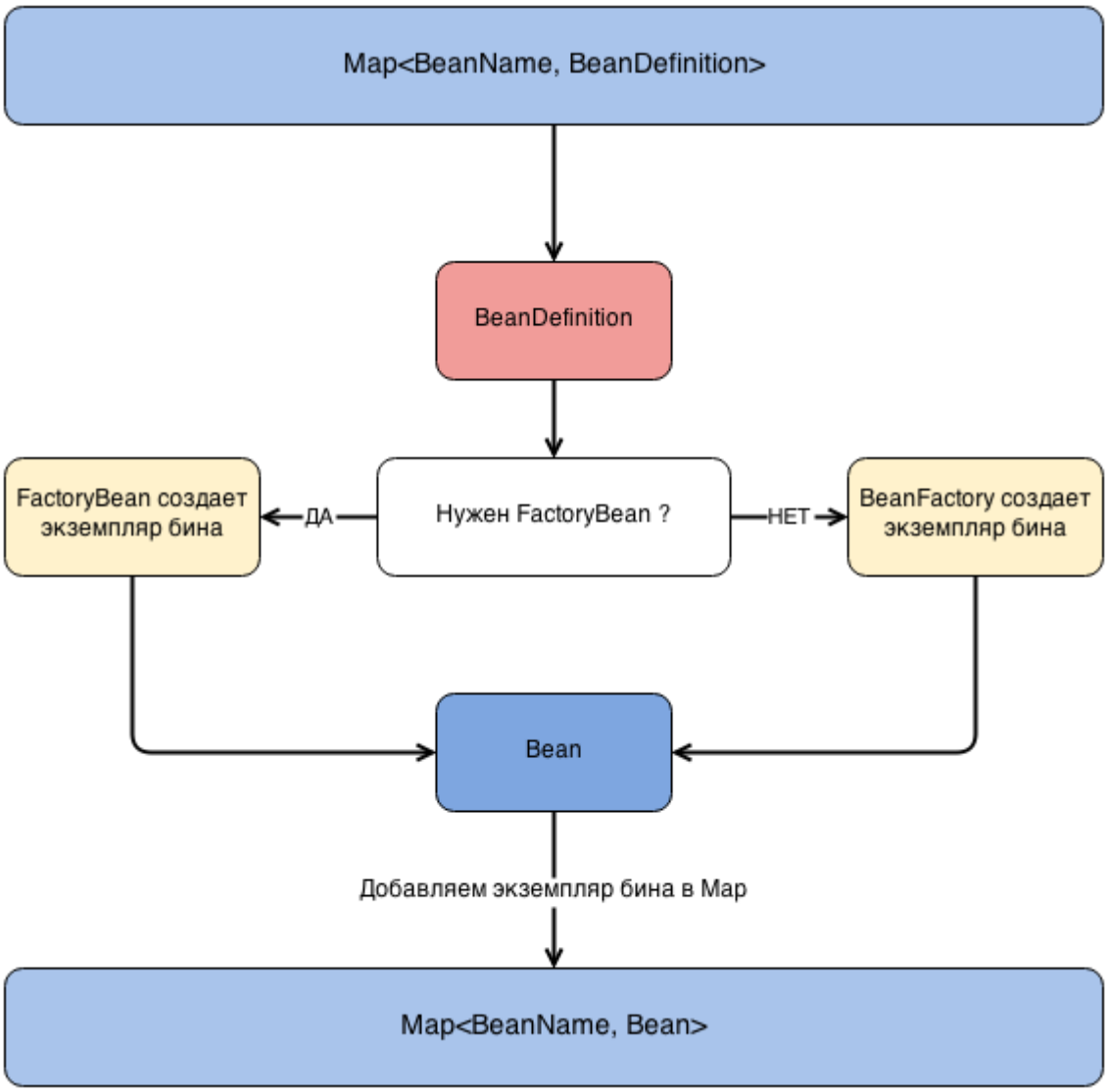
Добавим ее в xml и удалим объявленные до этого бины типа — *Color*.

```
<?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.sp
ringframework.org/schema/beans/spring-beans.xsd http://www.springfrang-context.xsd">

        <bean id="colorFactory" class="com.malahov.temp.ColorFactory"></bean>

</beans>
```

Теперь создание бина типа *Color.class* будет делегироваться ColorFactory, у которого при каждом создании нового бина будет вызываться метод **getObject**. Для тех кто пользуется JavaConfig, этот интерфейс будет абсолютно бесполезен.

Создание экземпляров бинов

Созданием экземпляров бинов занимается *BeanFactory* при этом, если нужно, делегирует это кастомным *FactoryBean*. Экземпляры бинов создаются на основе ранее созданных *BeanDefinition*.



Настройка созданных бинов

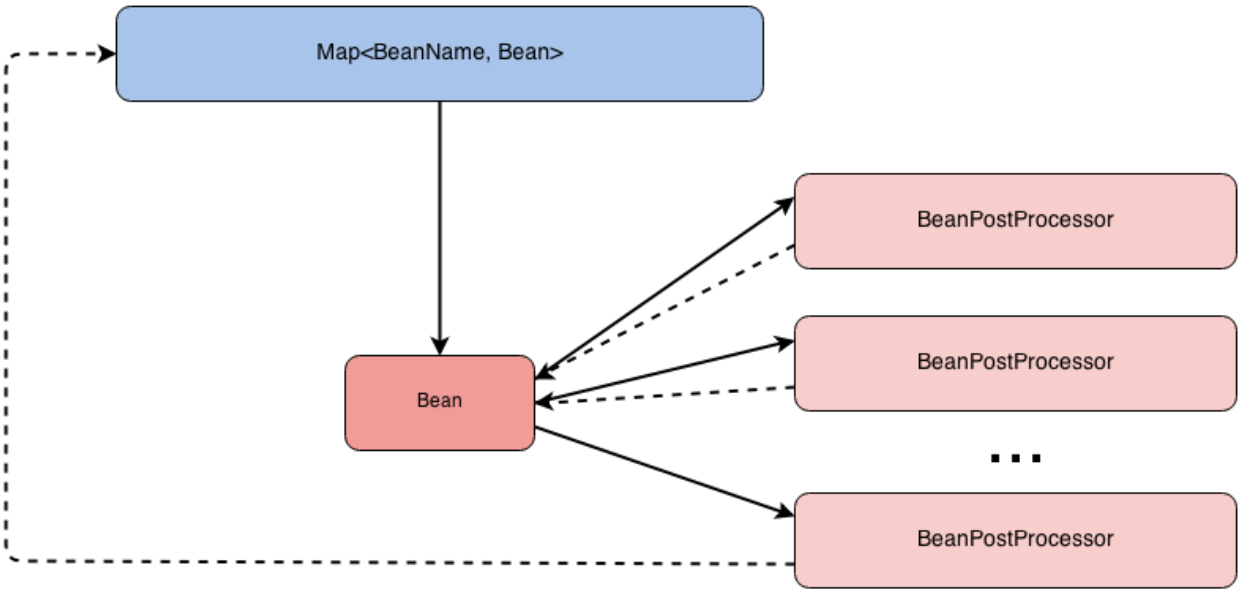
Интерфейс *BeanPostProcessor* позволяет вклиниться в процесс настройки ваших бинов до того, как они попадут в контейнер. Интерфейс несет в себе несколько методов.

```
public interface BeanPostProcessor {
        Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException;
        Object postProcessAfterInitialization(Object bean, String beanName) throws B
eansException;
}
```

Оба метода вызываются для каждого бина. У обоих методов параметры абсолютно одинаковые. Разница только в порядке их вызова. Первый вызывается до init-метода, второй, после. Важно понимать, что на данном этапе экземпляр бина уже создан и идет его донастройка. Тут есть два важных момента:

1. Оба метода в итоге должны вернуть бин. Если в методе вы вернете null, то при получении этого бина из контекста вы получите null, а поскольку через бинпостпроцессор проходят все бины, после поднятия контекста, при запросе любого бина вы будете получать null.
2. Если вы хотите сделать прокси над вашим объектом, то это принято делать после вызова init метода, иначе говоря это нужно делать в методе **postProcessAfterInitialization**.

Процесс донастройки показан на рисунке ниже. Порядок в котором будут вызваны *BeanPostProcessor* не известен, но мы точно знаем что выполнены они будут последовательно.



Итак:
Интерфейс BeanPostProcessor имеет всего два метода:
– postProcessBeforeInitialization
– postProcessAfterInitialization

Они позволяют разработчику самому имплементировать некоторые методы бинов перед инициализацией и после уничтожения экземпляров бина.

Данный интерфейс работает с экземплярами бинов, а это означает, что Spring IoC создаёт экземпляр бина, а затем BeanPostProcessor с ним работает.

ApplicationContext автоматически обнаруживает любые bean-компоненты, определенные в метаданных конфигурации, которые реализуют интерфейс BeanPostProcessor. Он регистрирует эти бины как постпроцессоры, чтобы их можно было вызывать позже при создании бина.

Затем Spring передаст каждый экземпляр bean-компонента этим двум методам до и после вызова метода обратного вызова инициализации, где вы можете обработать экземпляр bean-компонента так, как вам нравится.

```
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.stereotype.Component;

@Component
public class MyBeanPostProcessor implements BeanPostProcessor {

   @Override
   public Object postProcessAfterInitialization(Object object, String name) throws
BeansException {
      System.err.println("postProcessAfterInitialization(): " + object);
      return object;
   }

   @Override
   public Object postProcessBeforeInitialization(Object object, String name) throws
BeansException {
      return object;
   }

}
```

Запустите приложение

```
Output
postProcessAfterInitialization(): org.springframework.util.AntPathMatcher@6d
postProcessAfterInitialization(): org.springframework.web.util
  .UrlPathHelper@2fb76873
postProcessAfterInitialization(): ResourceHttpRequestHandler ["/resources/"]
postProcessAfterInitialization(): org.springframework.web.servlet.handler
  .SimpleUrlHandlerMapping@23f7d2f9
postProcessAfterInitialization(): by.pnv.controlles.HomeController@14ee34f9
postProcessAfterInitialization(): by.pnv.controlles.LoginController@63ea96e0
postProcessAfterInitialization(): org.springframework.web.servlet.view
  .InternalResourceViewResolver@4e9b7f2e
postProcessAfterInitialization(): org.springframework.web.servlet.i18n
  .AcceptHeaderLocaleResolver@71627391
postProcessAfterInitialization(): org.springframework.web.servlet.theme
  .FixedThemeResolver@2257a55a
postProcessAfterInitialization(): org.springframework.web.servlet.view
  .DefaultRequestToViewNameTranslator@3d4d0a0b
postProcessAfterInitialization(): org.springframework.web.servlet.support
  .SessionFlashMapManager@6a62000e
```

Гибридная конфигурация

С принятием версии 3.0 API сервлетов файл web.xml стал необязательным, и теперь мы можем использовать Java для настройки DispatcherServlet.

Мы можем зарегистрировать сервлет, реализующий WebApplicationInitializer. Это эквивалент конфигурации XML выше:

Для конфигурирования создайте пакет config и в нем класс

```
public class WebAppInitializer implements WebApplicationInitializer {


    @Override
    public void onStartup(ServletContext container) {
        XmlWebApplicationContext context = new XmlWebApplicationContext();
        context.setConfigLocation("/WEB-INF/spring/servlet-context.xml");

        ServletRegistration.Dynamic dispatcher = container
                .addServlet("dispatcher", new DispatcherServlet(context));

        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");
    }
}
```

Здесь идет реализация интерфейса WebApplicationInitializer

Переопределяя метод onStartup, мы создаем новый XmlWebApplicationContext, настроенный с тем же файлом, переданным в качестве contextConfigLocation сервлету в примере XML Затем мы создаем экземпляр DispatcherServlet с новым контекстом, который мы только что создали. И, наконец, мы регистрируем сервлет с шаблоном сопоставления URL

Поэтому мы использовали Java, чтобы объявить сервлет и связать его с отображением URL, но мы сохранили конфигурацию в отдельном файле XML: `servlet-context.xml`

Spring прочитает информацию конфигурации в этом классе, чтобы инициализировать (initial) ваше веб приложение. Обычно в данном классе вы можете зарегистрировать **Servlet**, **Servlet Filter**, и **Servlet Listener** вместо того, чтобы регистрировать их в **web.xml**.

100% Java Конфигурация

На этот раз мы будем использовать контекст, основанный на аннотациях, чтобы мы могли использовать Java и аннотации для конфигурации и убрать необходимость в файлах XML, таких как dispatcher-config.xml, `servlet-context.xml`

```
public class WebAppInitializer implements WebApplicationInitializer {

    public void onStartup(ServletContext servletContext) throws ServletException {
        AnnotationConfigWebApplicationContext context = new
AnnotationConfigWebApplicationContext();
        context.register(WebConfig.class);
     //   context.register(SecurityConfig.class);
        context.setServletContext(servletContext);

        ServletRegistration.Dynamic dispatcher = servletContext.addServlet("dispatcher",
new DispatcherServlet(context));
        dispatcher.addMapping("/*");
        dispatcher.setLoadOnStartup(1);
    }
```

Этот тип контекста затем может быть настроен путем регистрации класса конфигурации:

```
context.register(WebConfig.class);
```

Или настройку всего пакета, который будет проверяться на наличие классов конфигурации:

```
context.setConfigLocation("by.pnv.config");
```

Следующим шагом является создание и регистрация нашего сервлета диспетчера:

```
ServletRegistration.Dynamic dispatcher = servletContext.addServlet("dispatcher", new
DispatcherServlet(context));
    dispatcher.addMapping("/*");
    dispatcher.setLoadOnStartup(1);
```

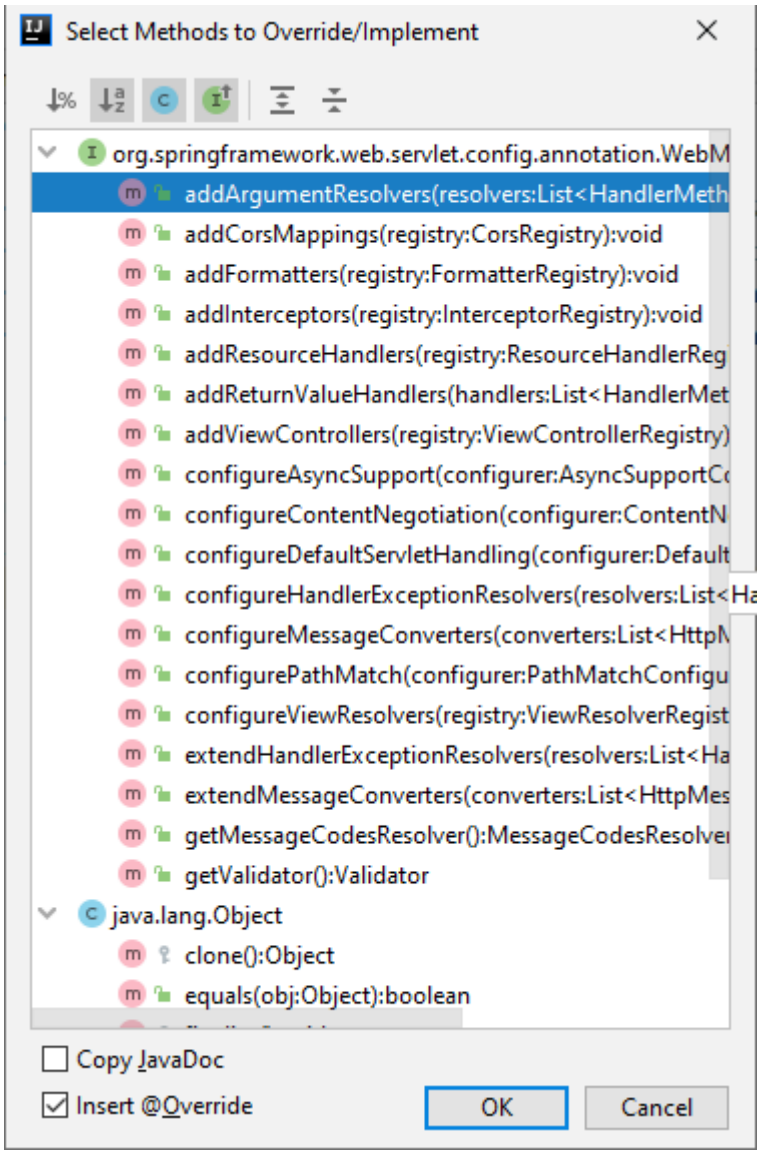
Конфигурация java и аннотации предлагает много преимуществ. Но это не всегда предпочтительный или даже возможный способ.

Класс конфигурации

Чтобы включить поддержку Spring MVC через класс конфигурации Java, все, что нам нужно сделать, это создать класс, например, **WebConfig** аннотированный @Configuration со следующим содержимым:

```
@Configuration
@EnableWebMvc
@ComponentScan({"by.pnv.config","by.pnv"})
public class WebConfig implements WebMvcConfigurer {

}
```

@Configuration сообщает Spring что данный класс является конфигурационным. Класс **WebConfig** реализует интерфейс **WebMvcConfigurer**, у которого есть целая куча методов, и настывает все по своему вкусу.



@EnableWebMvc — добавление этой аннотации к классу импортирует конфигурацию Spring MVC из WebMvcConfigurationSupport.

@ComponentScan сообщает Spring где искать компоненты, которыми он должен управлять, т.е. классы, помеченные аннотацией @Component или ее производными, такими как @Controller, @Repository, @Service. Эти аннотации автоматически определяют бин класса.

Первый метод

```
@Bean
public ViewResolver viewResolver() {
    InternalResourceViewResolver bean = new
InternalResourceViewResolver();

    bean.setViewClass(JstlView.class);
    bean.setPrefix("/WEB-INF/view/");
    bean.setSuffix(".jsp");

    return bean;
}
```

Чтобы загрузить приложение, которое загружает эту конфигурацию, у нас есть класс инициализатора.

```
public class WebAppInitializer implements WebApplicationInitializer {
```

Обратите внимание, что для версий более ранних, чем Spring 5, мы должны использовать класс *WebMvcConfigurerAdapter* вместо интерфейса.

Мы определяем автоматический контроллер с помощью метода addViewController ()

```
@Override
public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("index");
}
```

Метод addViewControllers () получает ViewControllerRegistry, который можно использовать для регистрации одного или нескольких контроллеров представления. Вызываем addViewController (), передавая "/", то есть путь, по которому контроллер представления будет обрабатывать запросы GET. Этот метод возвращает объект ViewControllerRegistration, в котором вызываем setViewName (), чтобы указать начальное представление, на которое должен быть перенаправлен запрос на «/».