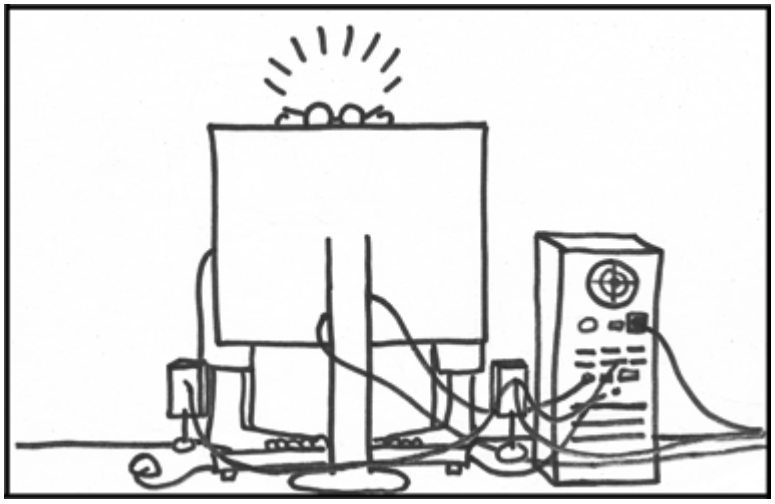
Делаем простое веб приложение на Spring Framework MVC



В данной статье я хочу рассказать начинающим Java разработчикам, как написать простое веб приложение, используя популярный фреймворк *Spring Framework.*

При разрабокте приложения мы будем использовать утилиту *Ant* для автоматизации действий и изучим, как писать простой тест с помощью библиотеки *JUnit*. Весь код будем писать в *Eclipse IDE*.

Статья написана на основе первой части руководства *«Introduction to Spring MVC»*. Вам достаточно иметь лишь общее представление о Spring, чтобы прочитать статью с пользой.

Так что милости просим :)

Для конфигурирования приложения в Spring можно использовать несколько спобов. Наиболее популярный путь – *вынесение конфигурации в xml-файлы*. Также это наиболее традиционный путь, который используется в фреймворке с первого релиза. С введением *аннотаций* в языке Java 5 появилась возможность настраивать фреймворк с помощью них (с версии 2.5). В данной статье мы будем использовать традиционный XML-стиль.

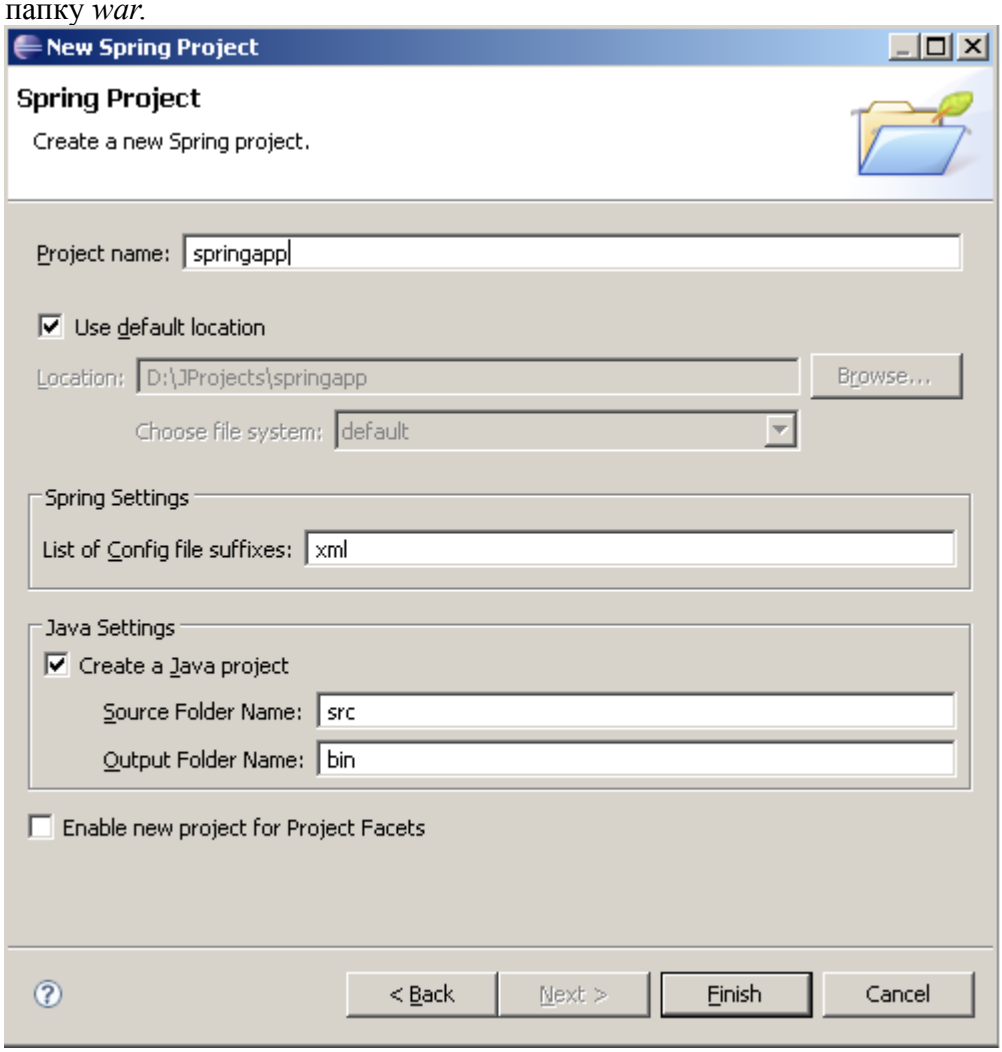
Необходимые инструменты

- Java SDK 1.5,
- Apache Tomcat 6,
- Ant 1.7,
- Junit 4.x,
- Eclipse 3.

Важное замечание: при копировании кода замените кавычки «елочки» на обыкновенные. Это избавит Ваш код от ошибок :)

1. Создание структуры проекта

На установленный Eclipse поставим плагин Spring IDE. Создадим Spring-проект *springapp*, добавим папку *war*.



2. Создадим index.jsp

springapp/war/index.jsp

```
<html>
   <head>
      <title>Example :: Spring Application</title>
   </head>
   <body>
      <h1>Example - Spring Application</h1>
      <p>This is my test.</p>
   </body>
</html>
```

В папке *war* создадим *WEB-INF*, и разместим в ней *web.xml*.

springapp/war/WEB-INF/web.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<web-app version='2.4'
    xmlns='java.sun.com/xml/ns/j2ee'
    xmlns:xsi='www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://java.sun.com/xml/ns/j2ee
    java.sun.com/xml/ns/j2ee/web-app_2_4.xsd' >
    <welcome-file-list>
      <welcome-file>
        index.jsp
      </welcome-file>
    </welcome-file-list>
</web-app>
```

3. Деплоим приложение в Tomcat

Для развертывания приложения на сервере воспользуемся ant-скриптом (для начала работы с Ant достаточно прочитать заметку о нем в Википедии). Скрипт будет содержать цели для компиляции, построения и переноса приложения.

springapp/build.xml

```
<?xml version='1.0'?>
<project name='springapp' basedir='.' default='usage'>
    <property file='build.properties' />
    <property name='src.dir' value='src' />
    <property name='web.dir' value='war' />
    <property name='build.dir' value='${web.dir}/WEB-INF/classes' />
    <property name='name' value='springapp' />
    <path id='master-classpath'>
      <fileset dir='${web.dir}/WEB-INF/lib'>
        <include name='*.jar' />
      </fileset>
      <!-- We need the servlet API classes: -->
      <!-- * for Tomcat 5/6 use servlet-api.jar -->
      <!-- * for other app servers - check the docs -->
      <fileset dir='${appserver.lib}'>
        <include name='servlet*.jar' />
      </fileset>
      <pathelement path='${build.dir}' />
    </path>
    <target name='usage'>
      <echo message='' />
      <echo message='${name} build file' />
      <echo message='-----------------------------------' />
      <echo message='' />
      <echo message='Available targets are:' />
      <echo message='' />
```

```
    <echo message='build --> Build the application' />
    <echo message='deploy --> Deploy application as directory' />
    <echo message='deploywar --> Deploy application as a WAR file' />
    <echo message='' />
  </target>
  <target name='build' description='Compile main source tree java files'>
    <mkdir dir='${build.dir}' />
    <javac destdir='${build.dir}' source='1.5' target='1.5' debug='true'
      deprecation='false' optimize='false' failonerror='true'>
      <src path='${src.dir}' />
      <classpath refid='master-classpath' />
    </javac>
  </target>
  <target name='deploy' depends='build' description='Deploy application'>
    <copy todir='${deploy.path}/${name}' preservelastmodified='true'>
      <fileset dir='${web.dir}'>
        <include name='**/*.*' />
      </fileset>
    </copy>
  </target>
  <target name='deploywar' depends='build'
    description='Deploy application as a WAR file'>
    <war destfile='${name}.war' webxml='${web.dir}/WEB-INF/web.xml'>
      <fileset dir='${web.dir}'>
        <include name='**/*.*' />
      </fileset>
    </war>
    <copy todir='${deploy.path}' preservelastmodified='true'>
      <fileset dir='.'>
        <include name='*.war' />
      </fileset>
    </copy>
  </target>
</project>
```

springapp/build.properties

```
# Ant properties for building the springapp
appserver.home=  C:/Program Files/Apache Software Foundation/Tomcat 6.0/

# for Tomcat 5 use $appserver.home}/server/lib
# for Tomcat 6 use $appserver.home}/lib
appserver.lib=${appserver.home}/lib
deploy.path=${appserver.home}/webapps
tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=tomcat
tomcat.manager.password=s3cret
```

Установите корректно переменную *appserver.home*. У меня она указывает на *C:/Program Files/Apache Software Foundation/Tomcat 6.0/*.

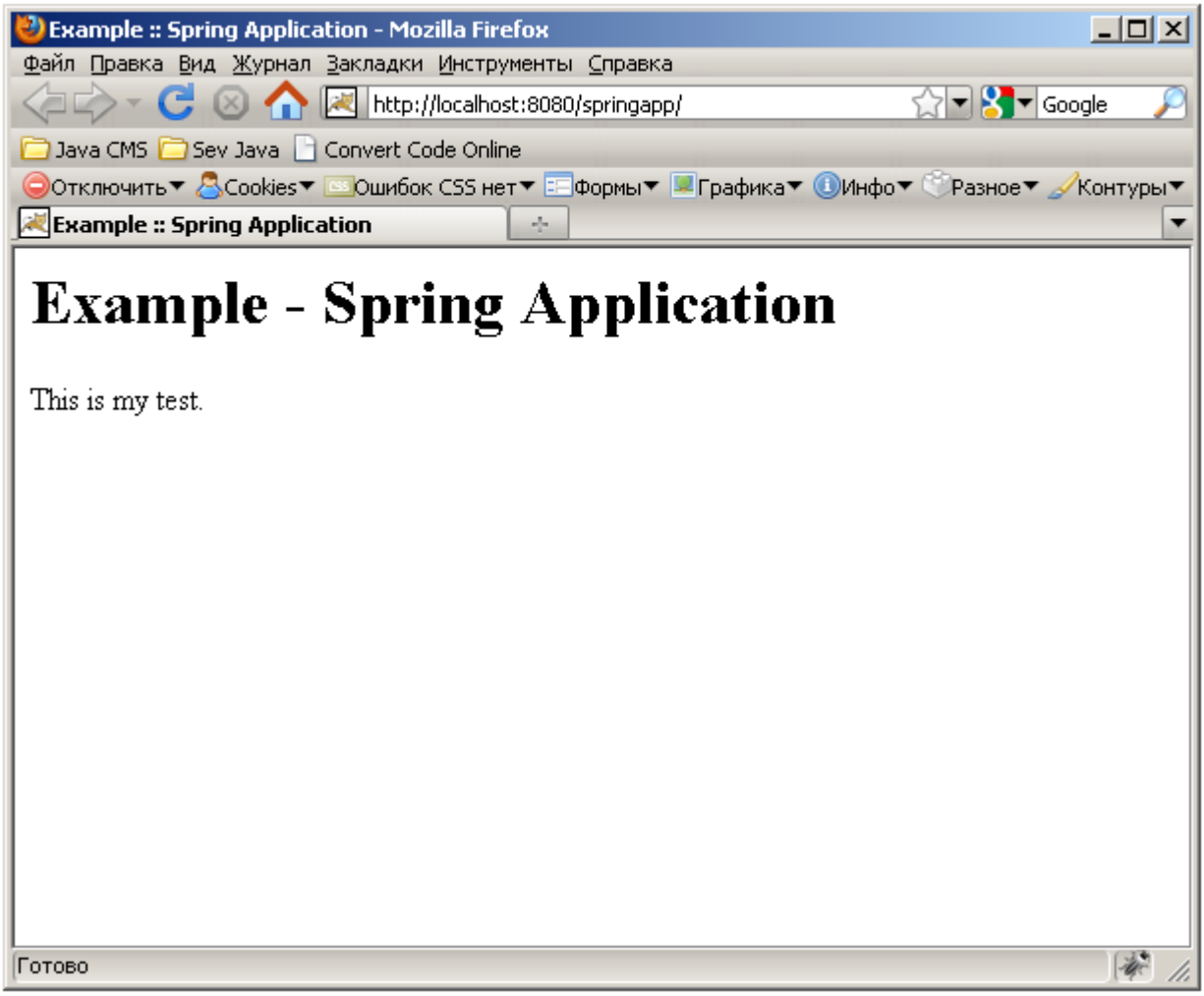
Для создания *пользователя Tomcat* в файле *appserver.home/conf/tomcat-users.xml* сделайте запись:

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="manager"/>
  <user username="tomcat" password="s3cret" roles="manager"/>
</tomcat-users>
```

Выполним build-скрипт: Контекстное меню файла build.xml > Run As > Ant Build > Выбрать цели build, deploy во вкладке Targets.

4. Проверим работоспособность приложения

Запустите Tomcat и откройте в браузере страницу localhost:8080/springapp/.



5. Скачиваем Spring Framework

Скачайте фреймворк, если вы ещё этого не сделали, и распакуйте его.

На этом настройка окружения закончена. Далее мы приступаем к конфигурированию самого приложения на Spring MVC.

6. Изменим web.xml в папке WEB-INF

Мы определим сервлет-дисптечер *DispatcherServlet* (также называемый *Front Controller*). Его цель – диспетчеризация поступающих запросов. Сделаем для этого сервлета *маппинг*. Мы решили все запросы с урлами вида '*.htm*' направлять на сервлет-дисптечер.

springapp/war/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="java.sun.com/xml/ns/j2ee"
   xmlns:xsi="www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="java.sun.com/xml/ns/j2ee
   java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
   <servlet>
      <servlet-name>springapp</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>springapp</servlet-name>
      <url-pattern>*.htm</url-pattern>
   </servlet-mapping>
   <welcome-file-list>
      <welcome-file>
         index.jsp
      </welcome-file>
   </welcome-file-list>
</web-app>
```

Создадим файл *springapp-servlet.xml*. Этот файл содержит *описания бинов* (Java-файлов), которые будет использовать DispatcherServlet. Иными словами файл определяет *контекст сервлета* (WebApplicationContext). По стандартному соглашению именования для Spring Web MVC, сервлет *springapp* будет иметь файл описания бинов с именем *springapp-servlet.xml*.

Добавим *описание бина* (bean entry) '/hello.htm' с классом *springapp.web.HelloController*. Эта запись определяет *Контроллер*, который будет использовать приложение для обслуживания запроса с урлом '*/hello.htm*'. Для маппинга урла на объект, что его будет обрабатывать, фреймворк Spring Web MVC использует класс, реализующий интерфейс *HandlerMapping*. По умолчанию для маппинга используется класс *BeanNameUrlHandlerMapping*.

В отличие от DispatcherServlet, *HelloController* ответственный за обработку запроса на конкретную страницу. Его также называют '*Page Controller*'.

springapp/war/WEB-INF/springapp-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"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
        <!-- the application context definition for the springapp
DispatcherServlet -->
        <bean name="/hello.htm" class="springapp.web.HelloController" />
</beans>
```

7. Скопируем библиотеки в 'WEB-INF/lib'

Создадим директорию *war/WEB-INF/lib* и скопируем в нее необходимые библиотеки Spring:

- spring.jar (из spring-framework-2.5/dist);
- spring-webmvc.jar (из spring-framework-2.5/dist/modules);
- commons-logging.jar (из spring-framework-2.5/lib/jakarta-commons);
- junit-4.4.jar (из spring-framework-2.5/lib/ junit).

Добавим в Eclipse-проект необходимые библиотеки:

- Контекстное меню проекта > Build Path > Configure Build Path > Add External JARs > 'war/WEB-INF/lib';
- Контекстное меню проекта > Build Path > Configure Build Path > Add Library > Server Runtime > Tomcat.

8. Создадим Контроллер

Создадим Контроллер *HelloController* в пакете *springapp.web*.

springapp/src/springapp/web/HelloController.java

```
package springapp.web;

import org.springframework.web.servlet.mvc.Controller;
import org.springframework.web.servlet.ModelAndView;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import java.io.IOException;

public class HelloController implements Controller {
  protected final Log logger = LogFactory.getLog(getClass());
```

```
    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        logger.info(«Returning hello view»);
        return new ModelAndView(«hello.jsp»);
    }
}
```

Это очень упрощенная реализация Контроллера. Мы будем её расширять в дальнейшем, и также будем использовать готовые реализации контроллеров из фреймворка.

В модели MVC Контроллер обрабатывает запрос и возвращает *Модель-и-Вид* (ModelAndView) – в нашем случае страницу '*hello.jsp*'. Модель, которую возвращает Контроллер, на самом деле разрешается через *ViewResolver*. Так как мы не указали явно ViewResolver, то будет использоваться резолвер по-умолчанию, который просто направляет запрос на адрес ресурса, указанный в Модели-и-Виде. В дальнейшем мы это изменим.

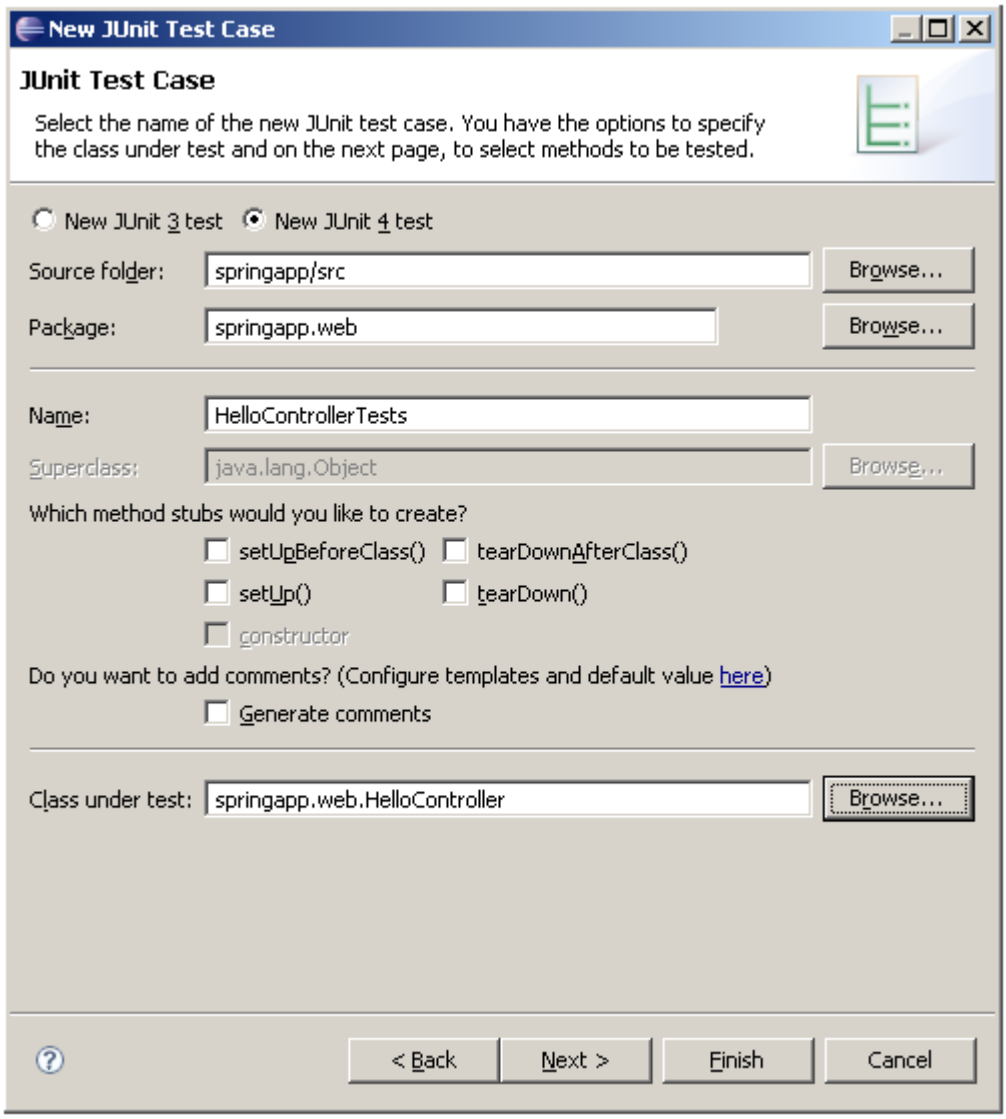
Также мы указали *логгер*, с помощью которого сможем проверить выполненную приложением работу. При использовании Tomcat журнал работы приложения мы сможем просмотреть в файле *catalina.out*, который можно найти по адресу *${Tomcat.home}/log*.

9. Напишем тест для Контроллера

Тестирование – это один из самых важных этапов разработки сложных программных систем. Также это основополагающая практика при *Agile software development*. Многие считают, что наилучшее время написания тестов – втечение разработки, а не после. Так что, несмотря на простоту разработанного нами контроллера, мы напишем к нему тест.

Создадим тест *HelloControllerTests*, расширяющий класс *TestCase* из библиотеки *Junit*: New > Other > JUnit > JUnit Test Case.

Это модульный тест (unit test). Он проверяет, совпадает ли имя Вида, возвращаемое через handleRequest() с Видом, которое мы ожидаем: '*hello.jsp*'.



springapp/src/springapp/web/HelloControllerTests.java

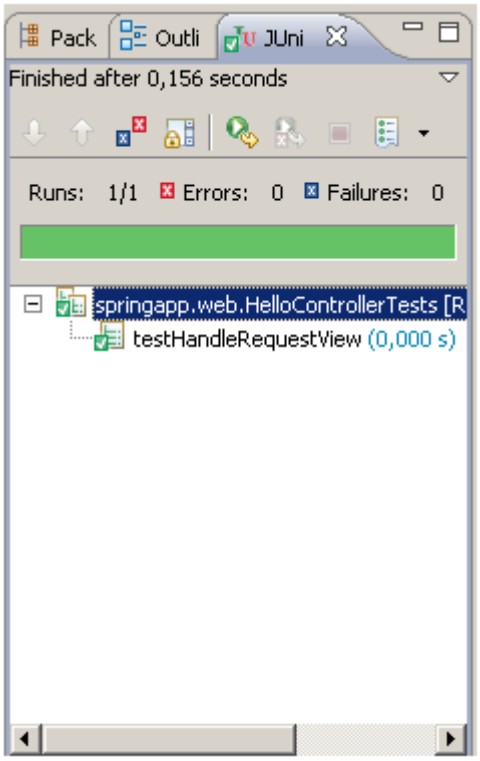
```
package springapp.web;

import org.springframework.web.servlet.ModelAndView;
import springapp.web.HelloController;
import static org.junit.Assert.*;
import org.junit.Test;

public class HelloControllerTests {

   @Test
   public void testHandleRequestView() throws Exception {
      HelloController controller = new HelloController();
      ModelAndView modelAndView = controller.handleRequest(null, null);
      assertEquals(«hello.jsp», modelAndView.getViewName());
   }
}
```

Для запуска теста используем меню: Run > Run As > JUnit Test.
Результат выполнения теста:



Ещё одной практикой Agile-разработки является непрерывная интеграция (Continuous Integration). Хорошей идеей будет запуск тестов с каждым билдом (build), для того, чтобы быть уверенным в правильном поведении своего кода (в идеальном варианте тесты запускаются автоматически при каждом билде).

10. Создадим Вид

Настало время создать *Вид*. В нашем случае это будет JSP-страница *hello.jsp*.

springapp/war/hello.jsp

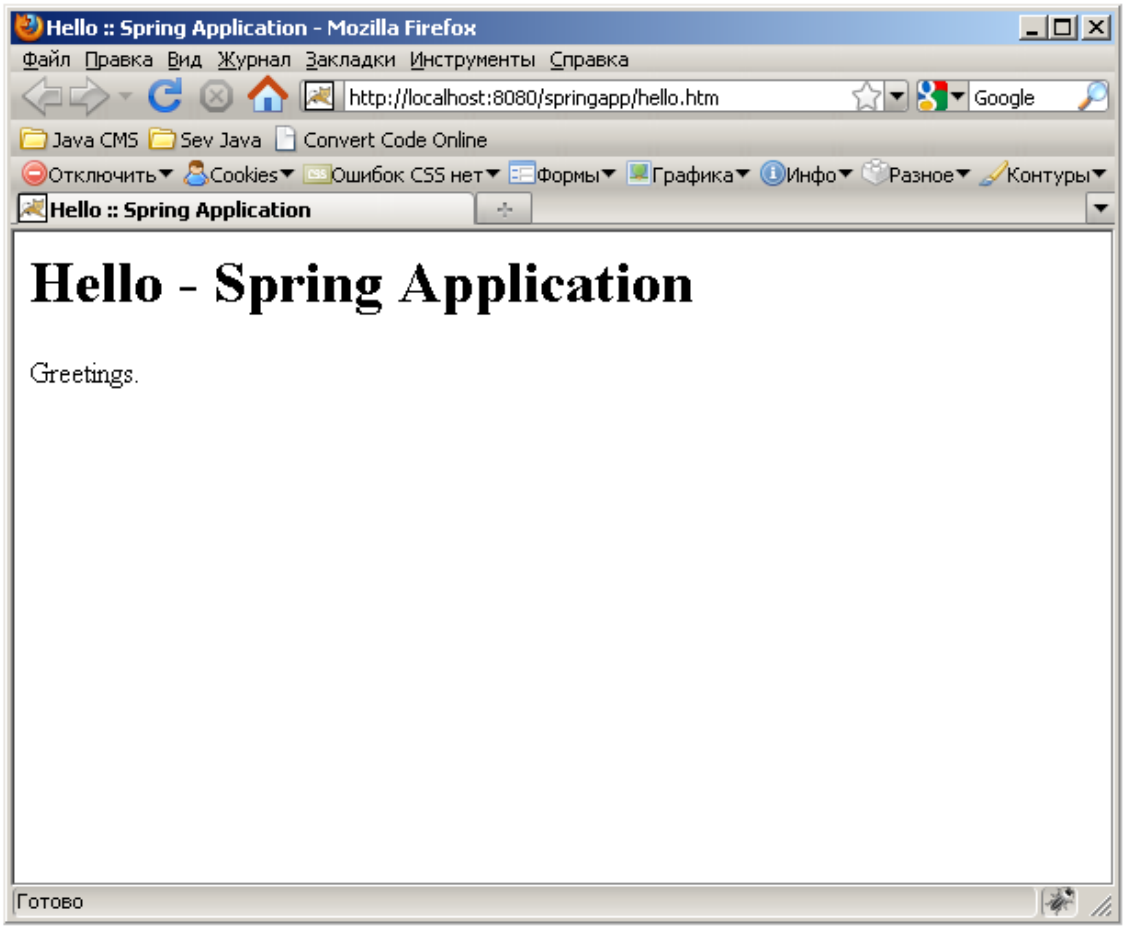
```
<html>
  <head>
    <title>Hello :: Spring Application</title>
  </head>
  <body>
    <h1>Hello - Spring Application</h1>
    <p>Greetings.</p>
  </body>
</html>
```

11. Скомпилируем и развернём приложение на сервере

Для файла *build.xml* добавим в *Classpath* библиотеку *springapp\war\WEB-INF\lib\junit-4.4.jar*, и выполним цели Build и Deploy (Контекстное меню файла build.xml > Run As > Ant Build… > вкладки Targets, Classpath).

12. Попробуем запустить приложение

В браузере наберем адрес http://localhost:8080/springapp/hello.htm.

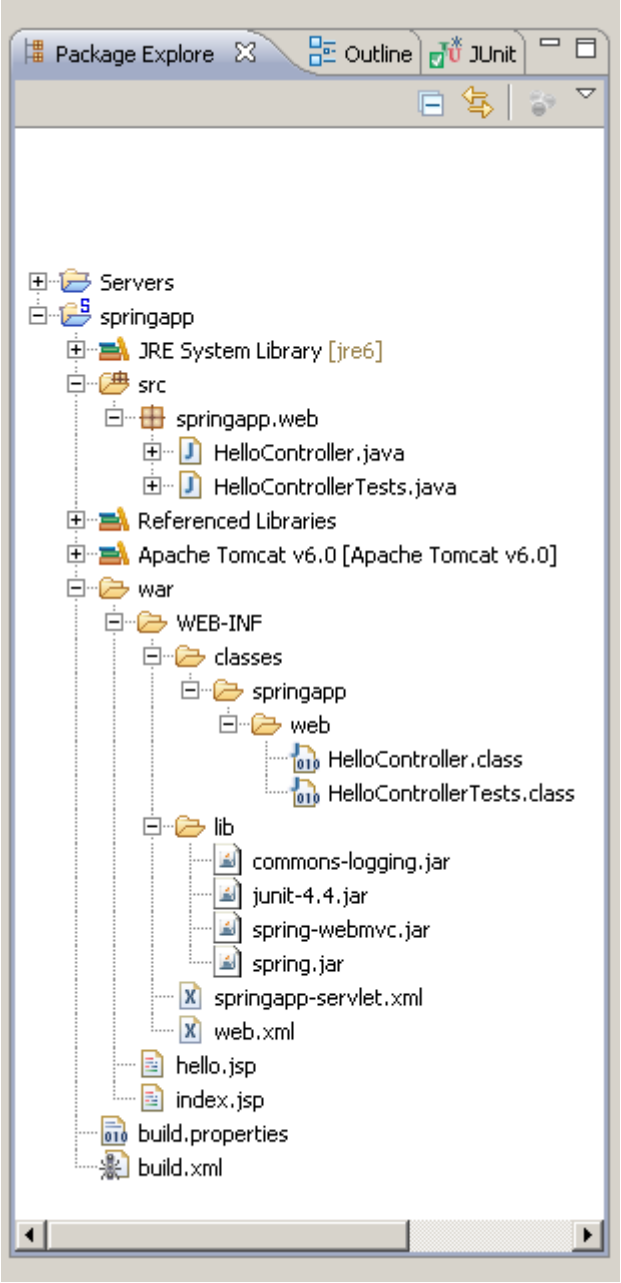


13. Резюме

Давайте бегло просмотрим, что было сделано.

1. *Стартовая страница приложения* – index.jsp. Служит для того, чтобы удостовериться в правильности установки окружения. Позднее мы ее изменим.
2. *Сервлет-диспетчер* (DispatcherServlet или Front controller) с соответствующим файлом описания springapp-servlet.xml.
3. *Контроллер* (Page controller) HelloController с базовой функциональностью – просто возвращает Модель-И-Вид. На данный момент Модель пустая. Полная реализация Модели будет сделана в дальнейшем.
4. *Модульный тест для контроллера* HelloControllerTests, проверяющий, соответствует ли имя вида ожидаемому.
5. *Вид* приложения – hello.jsp.

Ниже представлена структура проекта, которая должна быть после выполнения всех инструкций.



Готовый Eclipse-проект можно скачать здесь.

Спасибо за внимание. Желаю успехов!

Перевод подготовил: Антон Щастный (schastny.net). 2010 год.