Занятие № 17.

Spring Security.

Spring Security. Назначение.

Spring Security на практике

Spring Security. Назначение.

Spring Security это Java/JavaEE framework, предоставляющий механизмы построения систем аутентификации и авторизации, а также другие возможности обеспечения безопасности для корпоративных приложений, созданных с помощью Spring Framework. Проект был начат Беном Алексом (Ben Alex) в конце 2003 года под именем «Acegi Security», первый релиз вышел в 2004 году. Впоследствии проект был поглощён Spring'ом и стал его официальным дочерним проектом. Впервые публично представлен под новым именем Spring Security 2.0.0 в апреле 2008 года.

Ключевые объекты контекста Spring Security:

- SecurityContextHolder, в нем содержится информация о текущем контексте безопасности приложения, который включает в себя подробную информацию о пользователе(Principal) работающем в настоящее время с приложением. По умолчанию SecurityContextHolder используетThreadLocal для хранения такой информации, что означает, что контекст безопасности всегда доступен для методов исполняющихся в том же самом потоке. Для того что бы изменить стратегию хранения этой информации можно воспользоваться статическим методом класса SecurityContextHolder.setStrategyName(String strategy). Более подробно SecurityContextHolder
- SecurityContext, содержит объект Authentication и в случае необходимости информацию системы безопасности, связанную с запросом от пользователя.
- Authentication представляет пользователя (Principal) с точки зрения Spring Security.
- GrantedAuthority отражает разрешения выданные пользователю в масштабе всего приложения, такие разрешения (как правило называются «роли»), например ROLE_ANONYMOUS, ROLE_USER, ROLE_ADMIN.

- UserDetails предоставляет необходимую информацию для построения объекта Authentication из DAO объектов приложения или других источников данных системы безопасности. Объект UserDetailscодержит имя пользователя, пароль, флаги: isAccountNonExpired, isAccountNonLocked, isCredentialsNonExpired, isEnabled и Collection — прав (ролей) пользователя.
- UserDetailsService, используется чтобы создать UserDetails объект путем реализации единственного метода этого интерфейса

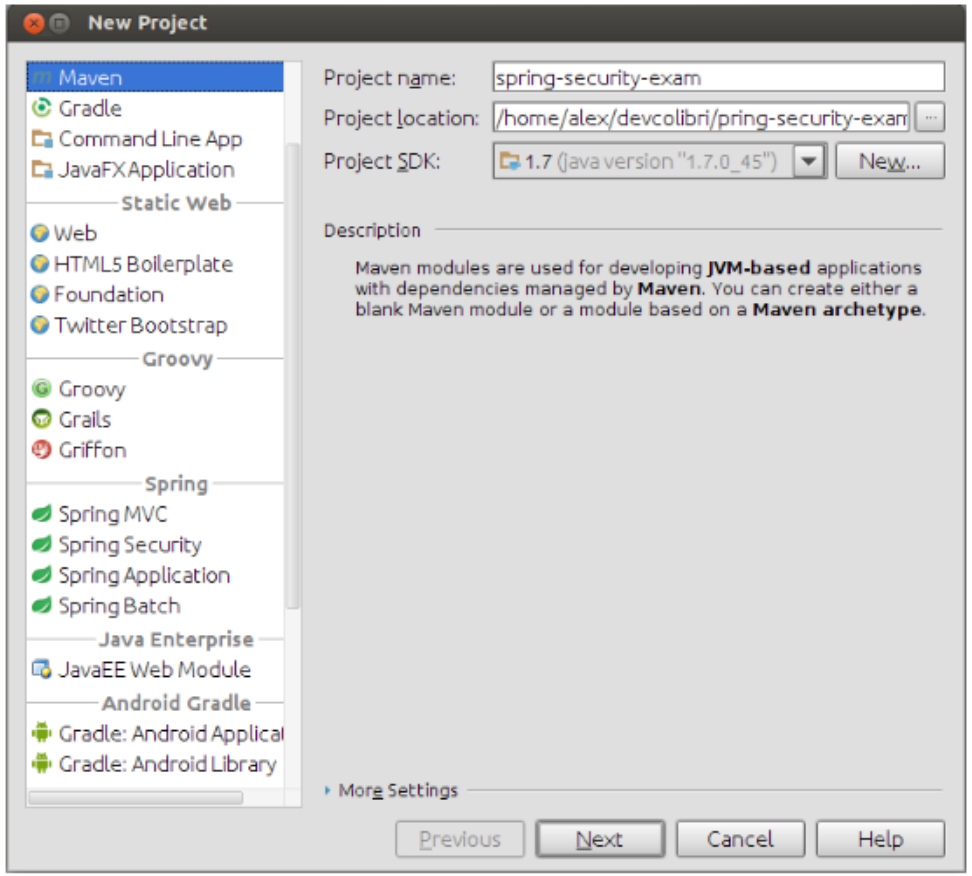
```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
```

Позволяет получить из источника данных объект пользователя и сформировать из него объект UserDetails, который будет использоваться контекстом Spring Security.

Spring Security на практике

Шаг 1. Создание проекта и подключение зависимостей

Начнем с традиционного создания проекта:



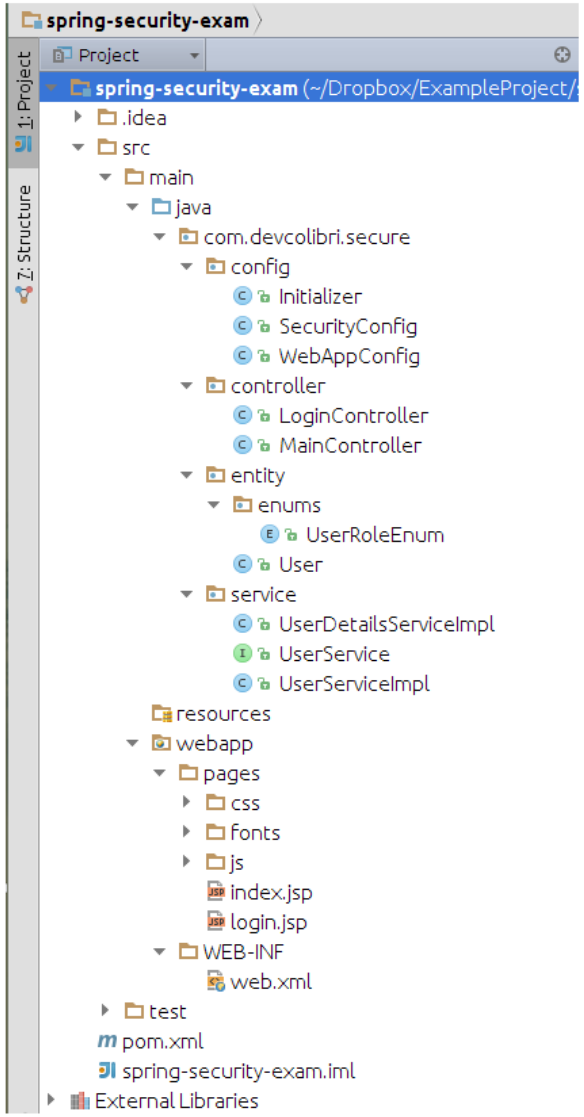
Создали Maven проект, теперь добавим зависимости в **pom.xml** их будет довольно таки много, поэтому прикладываю полный pom файл:

```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>spring-security-exam</groupId>
    <artifactId>spring-security-exam</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <properties>
        <spring.mvc>4.0.0.RELEASE</spring.mvc>
        <javax.servlet>3.0.1</javax.servlet>
        <jstl.version>1.2</jstl.version>
        <spring.securiry>3.2.0.RELEASE</spring.securiry>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring.mvc}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.mvc}</version>
        </dependency>

        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>${javax.servlet}</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>jstl</groupId>
            <artifactId>jstl</artifactId>
            <version>${jstl.version}</version>
        </dependency>

```

```
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-web</artifactId>
            <version>${spring.securiry}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-core</artifactId>
            <version>${spring.securiry}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-config</artifactId>
            <version>${spring.securiry}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-taglibs</artifactId>
            <version>${spring.securiry}</version>
        </dependency>

    </dependencies>

    <build>
        <finalName>secure-exam</finalName>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

По каждому из зависимостей не буду останавливаться, но включил самые обязательные зависимости для реализации Spring Security.

Сразу же покажу вам какая должна быть структура проекта, чтобы вы не терялись в создании классов и файлов.



Шаг 2. Начинаем с конца, делаем внешний вид

Для большей привликательности, как к внешнему виду так и мотивации изучить этот урок я решил использовать Bootstrap 3.

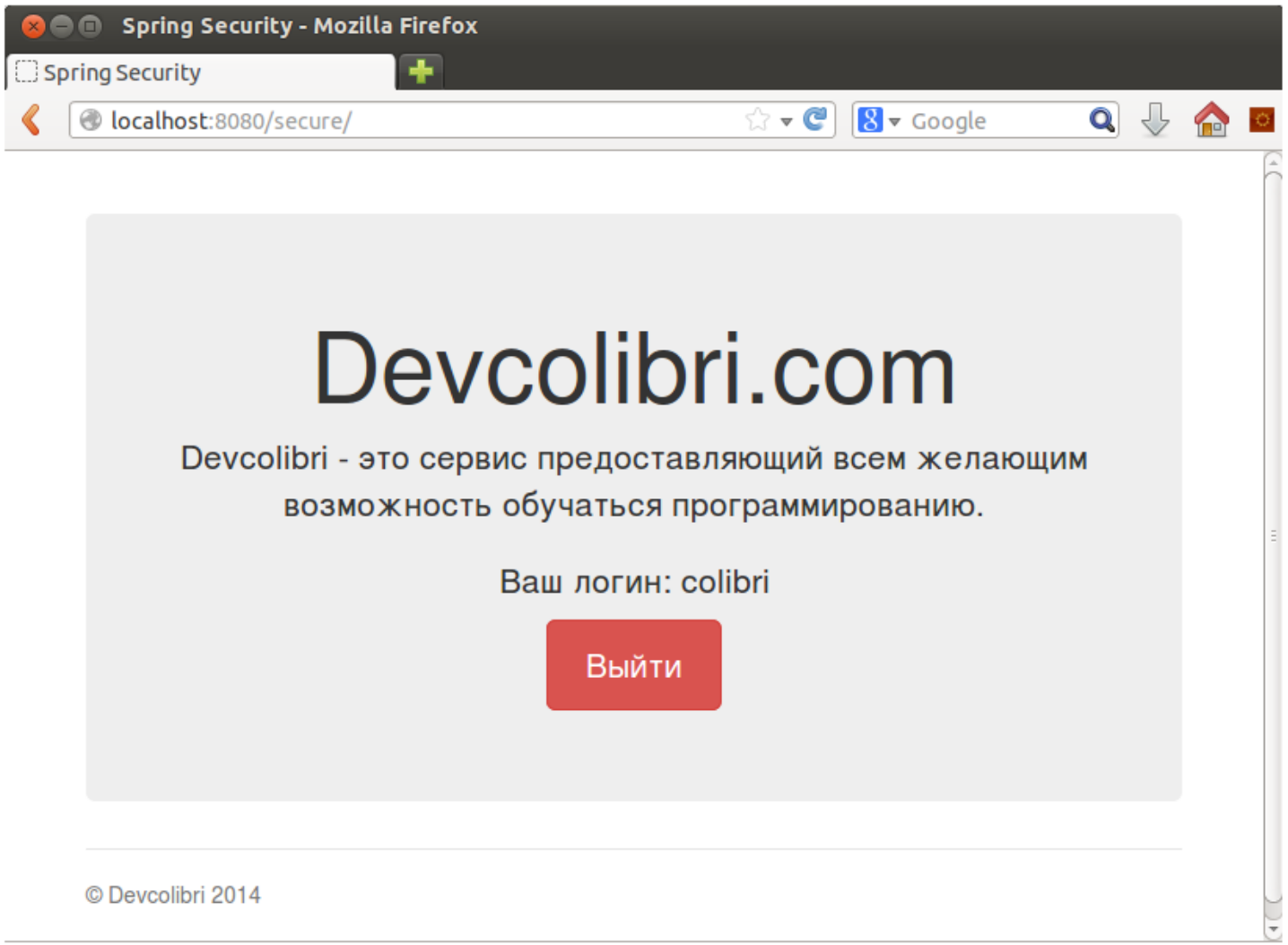
Скачиваем bootstrap и в проекте создаем папку **webapp/pages/** и копируем туда файлы bootstrap-а. (*смотрите на структуре проекта*)

Теперь создаем webapp/pages/**index.jsp:**

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec" %>

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description" content="">
    <meta name="author" content="">

    <title>Spring Security</title>

    <!-- Bootstrap core CSS -->
    <link href="<c:url value="/pages/css/bootstrap.css" />" rel="stylesheet">

    <!-- Custom styles for this template -->
    <link href="<c:url value="/pages/css/jumbotron-narrow.css" />" rel="stylesheet">

    <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media queries -->
    <!--[if lt IE 9]>
    <script src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>
    <script src="https://oss.maxcdn.com/libs/respond.js/1.3.0/respond.min.js"></script>
    <![endif]-->
</head>

<body>
```

```
28
29  <body>
30
31  <div class="container">
32
33      <div class="jumbotron" style="margin-top: 20px;">
34          <h1>Devcolibri.com</h1>
35          <p class="lead">
36              Devcolibri - это сервис предоставляющий всем желающим возможность обучаться программированию.
37          </p>
38          <sec:authorize access="!isAuthenticated()">
39              <p><a class="btn btn-lg btn-success" href="<c:url value="/login" />" role="button">Войти</a></p>
40          </sec:authorize>
41          <sec:authorize access="isAuthenticated()">
42              <p>Ваш логин: <sec:authentication property="principal.username" /></p>
43              <p><a class="btn btn-lg btn-danger" href="<c:url value="/logout" />" role="button">Выйти</a></p>
44
45          </sec:authorize>
46      </div>
47
48      <div class="footer">
49          <p>© Devcolibri 2014</p>
50      </div>
51
52  </div>
53  </body>
54  </html>
```

Выглядеть эта страница будет так:



Создаем следующую страницу на которой собственно и будет происходить вход на сайт webapp/pages/**login.jsp**:

```jsp
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description" content="">
    <meta name="author" content="">

    <title>Spring Security</title>

    <!-- Bootstrap core CSS -->
    <link href="<c:url value="/pages/css/bootstrap.css" />" rel="stylesheet">

    <!-- Custom styles for this template -->
    <link href="<c:url value="/pages/css/signin.css" />" rel="stylesheet">

    <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media queries -->
    <!--[if lt IE 9]>
    <script src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>
    <script src="https://oss.maxcdn.com/libs/respond.js/1.3.0/respond.min.js"></script>
    <![endif]-->
</head>

<body>

<div class="container" style="width: 300px;">
    <c:url value="/j_spring_security_check" var="loginUrl" />
    <form action="${loginUrl}" method="post">
        <h2 class="form-signin-heading">Please sign in</h2>
        <input type="text" class="form-control" name="j_username" placeholder="Email address" required autofocus valu
        <input type="password" class="form-control" name="j_password" placeholder="Password" required value="1234">
        <button class="btn btn-lg btn-primary btn-block" type="submit">Войти</button>
    </form>
</div>

</body>
</html>
```

Создаем следующую страницу на которой собственно и будет происходить вход на сайт webapp/pages/**login.jsp**:

```
e contentType="text/html; charset=UTF-8" language="java" %>
lib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

PE html>
ang="en">

ta charset="utf-8">
ta http-equiv="X-UA-Compatible" content="IE=edge">
ta name="viewport" content="width=device-width, initial-scale=1.0">
ta name="description" content="">
ta name="author" content="">

tle>Spring Security</title>

- Bootstrap core CSS -->
nk href="<c:url value="/pages/css/bootstrap.css" />" rel="stylesheet">

- Custom styles for this template -->
nk href="<c:url value="/pages/css/signin.css" />" rel="stylesheet">

- HTML5 shim and Respond.js IE8 support of HTML5 elements and media queries -->
-[if lt IE 9]>
ript src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>
ript src="https://oss.maxcdn.com/libs/respond.js/1.3.0/respond.min.js"></script>
endif]-->



ass="container" style="width: 300px;">
url value="/j_spring_security_check" var="loginUrl" />
rm action="${loginUrl}" method="post">
 <h2 class="form-signin-heading">Please sign in</h2>
 <input type="text" class="form-control" name="j_username" placeholder="Email address" required autofocus value="colibri">
 <input type="password" class="form-control" name="j_password" placeholder="Password" required value="1234">
 <button class="btn btn-lg btn-primary btn-block" type="submit">Войти</button>
orm>
```

Тут обратите особое внимание на обя тега input, а именно на их name:

name=»**j_username**»

name=»**j_password**»

В данном случае эти поля должны быть именно с такими значениями.

А также вы возможно уже увидели эту строку:

```
1 <c:url value="/j_spring_security_check" var="loginUrl" />
```

так мы создали переменную, которая будет хранить ссылку на security check, он выполняет аутентификация, где значение **value** обязательно должно быть таким.

Выглядеть она будет так:

Шаг 3. Создаем контроллеры

О создании MVC проекта на Spring можно почитать урок Spring MVC Hello World

Создаем пакет **controller** и класс **MainController**:

```
1  package com.devcolibri.secure.controller;
2
3  import org.springframework.stereotype.Controller;
4  import org.springframework.ui.Model;
5  import org.springframework.web.bind.annotation.RequestMapping;
6  import org.springframework.web.bind.annotation.RequestMethod;
7
8  @Controller
9  @RequestMapping("/")
10 public class MainController {
11
12     @RequestMapping(method = RequestMethod.GET)
13     public String start(Model model){
14         return "index";
15     }
16
17 }
```

Этот контроллер будет просто направлять пользователя на страницу **index.jsp.**

А теперь создадим второй аналогичный контроллер, который будет показывать пользователю страничку **login.jsp.**

Создаем класс и называем его **LoginController:**

```
1  package com.devcolibri.secure.controller;
2
3  import org.springframework.stereotype.Controller;
4  import org.springframework.ui.Model;
5  import org.springframework.web.bind.annotation.RequestMapping;
6  import org.springframework.web.bind.annotation.RequestMethod;
7
8  @Controller
9  @RequestMapping("/login")
10 public class LoginController {
11
12     @RequestMapping(method = RequestMethod.GET)
13     public String loginPage(Model model){
14         return "login";
15     }
16
17 }
```

Теперь мы сможем ходить по страничкам.

Шаг 4. Конфигурирование Spring MVC

Теперь нам нужно сконфигурировать Spring MVC чтобы он имел возможность разворачивать свой контекс и мы могли получать доступ к нашим бинам.

Создаем в пакете config класс **WebAppConfig** и наследуем его от WebMvcConfigurerAdapter:

```
1  package com.devcolibri.secure.config;
2
3  import com.devcolibri.secure.service.UserDetailsServiceImpl;
4  import org.springframework.context.annotation.Bean;
5  import org.springframework.context.annotation.ComponentScan;
6  import org.springframework.context.annotation.Configuration;
7  import org.springframework.web.servlet.config.annotation.EnableWebMvc;
8  import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
9  import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
10 import org.springframework.web.servlet.view.InternalResourceViewResolver;
11 import org.springframework.web.servlet.view.JstlView;
12 import org.springframework.web.servlet.view.UrlBasedViewResolver;
13
14 @Configuration
15 @EnableWebMvc
16 @ComponentScan("com.devcolibri")
17 public class WebAppConfig extends WebMvcConfigurerAdapter {
18
19     @Override
20     public void addResourceHandlers(ResourceHandlerRegistry registry) {
21         registry.addResourceHandler("/pages/**").addResourceLocations("/pages/");
22     }
23
24     @Bean
25     public UrlBasedViewResolver setupViewResolver() {
26         UrlBasedViewResolver resolver = new UrlBasedViewResolver();
27         resolver.setPrefix("/pages/");
28         resolver.setSuffix(".jsp");
29         resolver.setViewClass(JstlView.class);
30
31         return resolver;
32     }
33
34 }
```

Шаг 5. Создание Entity

Создаем entity по сути это простой класс, так как мы не используем не ORM фреймворков, назовем его User:

```
package com.devcolibri.secure.entity;

public class User {

    private String login;
    private String password;

    public User(String login, String password) {
        this.login = login;
        this.password = password;
    }

    public User() {

    }

    public String getLogin() {
        return login;
    }

    public void setLogin(String login) {
        this.login = login;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

В будущем вы можите этот объект пополнять своими свойствами, но для примера нам достаточно знать **Login** (*логин*) и **Password** (*пароль*).

И еще в этом же пакете создадим пакет enums и в нем создаем enum **UserRoleEnum,** который будет определять роли пользователя:

```
package com.devcolibri.secure.entity.enums;

public enum UserRoleEnum {

    ADMIN,
    USER,
    ANONYMOUS;

    UserRoleEnum() {
    }

}
```

Теперь у нас есть 3 роли, которые мы немного позже будем использовать.

Шаг 6. Создаем слой Services

Обычно проекты имеют несколько уровней слоёв, о которых я еще напишу статью, но один из этих слоёв мы реализуем прям сейчас.

Нам нужно реальзовать Service tier (*сервис слой либо уровень обслуживания*). Создаем пакет service и в нем создаем интерфейс **UserService**:

```
package com.devcolibri.secure.service;

import com.devcolibri.secure.entity.User;

public interface UserService {

    User getUser(String login);

}
```

Этот сервисный интерфейс говорит о том, что у нас будет сервис который позволит получать User не важно как и откуда, но он позволит нам это.

А теперь давайте напишем первую реализацию данного интерфейса, для этого создаем на том же уровне класс **UserServiceImpl**:

```
package com.devcolibri.secure.service;

import com.devcolibri.secure.entity.User;
import org.springframework.stereotype.Service;

@Service
public class UserServiceImpl implements UserService {

    @Override
    public User getUser(String login) {
        User user = new User();
        user.setLogin(login);
        user.setPassword("7110eda4d09e062aa5e4a390b0a572ac0d2c0220");

        return user;
    }

}
```

Как видите реализация довольно простая, тут мы просто заполняем объект **User** используя **setters**, но мы также можем в этом севис методе вызвать метод из DAO, который бы достал нам этого юзера с БД например либо получил бы его с Web Service.

Обратите внимание, что я установил в поле Password специфичный пароль а именно зашифрованый в **SHA1** формате.

То есть я взял пароль «**1234**» и зашифровал его в **SHA1** формат с помощью online сервиса SHA1 online generator и получил уже зашифрованый пароль «**7110eda4d09e062aa5e4a390b0a572ac0d2c0220**«.

http://www.sha1-online.com

Шаг 7. Создаем реализацию UserDetailsService

Для того, чтобы связать наш сервис UserService со Spring Security, нам нужно реализовать специальный интерфейс фреймворка Spring Security который позволит выполнять аутентификацию пользователя на основании данных полученых с UserService написанного высше.

Создаем класс **UserDetailsServiceImpl** и реализуем его от UserDetailsService:

```java
package com.devcolibri.secure.service;

import com.devcolibri.secure.entity.User;
import com.devcolibri.secure.entity.enums.UserRoleEnum;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import java.util.HashSet;
import java.util.Set;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UserService userService;

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        // с помощью нашего сервиса UserService получаем User
        User user = userService.getUser("colibri");
        // указываем роли для этого пользователя
        Set<GrantedAuthority> roles = new HashSet();
        roles.add(new SimpleGrantedAuthority(UserRoleEnum.USER.name()));

        // на основании полученныйх даных формируем объект UserDetails
        // который позволит проверить введеный пользователем логин и пароль
        // и уже потом аутентифицировать пользователя
        UserDetails userDetails =
                new org.springframework.security.core.userdetails.User(user.getLogin(),
                                                                       user.getPassword(),
                                                                       roles);

        return userDetails;
    }

}
```

Этот сервис и является основной логикой аутентификаци с использованием Spring Security.

Шаг 8. Добавляем инициализацию бина UserDetailsServiceImpl в конфигурацию WebAppConfig

Для того чтобы наша реализация интерфейса UserDetailsService смогла инициализыроваться контейнером Spring нам нужно добавить его в WebAppConfig:

```
@Bean
public UserDetailsService getUserDetailsService(){
    return new UserDetailsServiceImpl();
}
```

После этого Spring контейнер будет знать какую реализацию интерфейса UserDetailsService нужно брать.

Шаг 9. Конфигурирование Spring Security

Теперь самое главное, заставить все это заработать :)

Создаем в пакете config класс **SecurityConfig** и наследуем его от WebSecurityConfigurerAdapter:

```
package com.devcolibri.secure.config;

import com.devcolibri.secure.service.UserDetailsServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.encoding.ShaPasswordEncoder;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserDetailsServiceImpl userDetailsService;

    // регистрируем нашу реализацию UserDetailsService
    // а также PasswordEncoder для приведения пароля в формат SHA1
    @Autowired
    public void registerGlobalAuthentication(AuthenticationManagerBuilder auth) throws Exception {
        auth
                .userDetailsService(userDetailsService)
                .passwordEncoder(getShaPasswordEncoder());
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
```

```
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // включаем защиту от CSRF атак
        http.csrf()
                .disable()
                // указываем правила запросов
                // по которым будет определятся доступ к ресурсам и остальным данным
                .authorizeRequests()
                .antMatchers("/resources/**", "/**").permitAll()
                .anyRequest().permitAll()
                .and();

        http.formLogin()
                // указываем страницу с формой логина
                .loginPage("/login")
                // указываем action с формы логина
                .loginProcessingUrl("/j_spring_security_check")
                // указываем URL при неудачном логине
                .failureUrl("/login?error")
                // Указываем параметры логина и пароля с формы логина
                .usernameParameter("j_username")
                .passwordParameter("j_password")
                // даем доступ к форме логина всем
                .permitAll();

        http.logout()
                // разрешаем делать логаут всем
                .permitAll()
                // указываем URL логаута
                .logoutUrl("/logout")
                // указываем URL при удачном логауте
                .logoutSuccessUrl("/login?logout")
                // делаем не валидной текущую сессию
                .invalidateHttpSession(true);

    }

    // Указываем Spring контейнеру, что надо инициализировать <b></b>ShaPasswordEncoder
    // Это можно вынести в WebAppConfig, но для понимаемости оставил тут
    @Bean
    public ShaPasswordEncoder getShaPasswordEncoder(){
        return new ShaPasswordEncoder();
    }

}
```

Это базовая конфигурация, которая нужна для наших нужд, она может расширятся.

Шаг 10. Регистрация конфигураций в контексте Spring

Чтобы Spring видел наши конфигурации, а именно WebAppConfig и SecurityConfig их нужно зарегистрировать в контексте Spring.

Создаем в пакете config класс **Initializer** и реализуем WebApplicationInitializer:

```java
package com.devcolibri.secure.config;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.ContextLoaderListener;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration.Dynamic;

public class Initializer implements WebApplicationInitializer {

    private static final String DISPATCHER_SERVLET_NAME = "dispatcher";

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();
        // регистрация конфигураций в Spring контексте
        ctx.register(WebAppConfig.class);
        ctx.register(SecurityConfig.class);
        servletContext.addListener(new ContextLoaderListener(ctx));

        ctx.setServletContext(servletContext);

        Dynamic servlet = servletContext.addServlet(DISPATCHER_SERVLET_NAME, new DispatcherServlet(ctx));
        servlet.addMapping("/");
        servlet.setLoadOnStartup(1);
    }
}
```

На этом этап конфигурации проекта можно щитать законченым.

В **web.xml** нужно добавить фильтр, который будет определять наши реквесты и проверять валидность сессии:

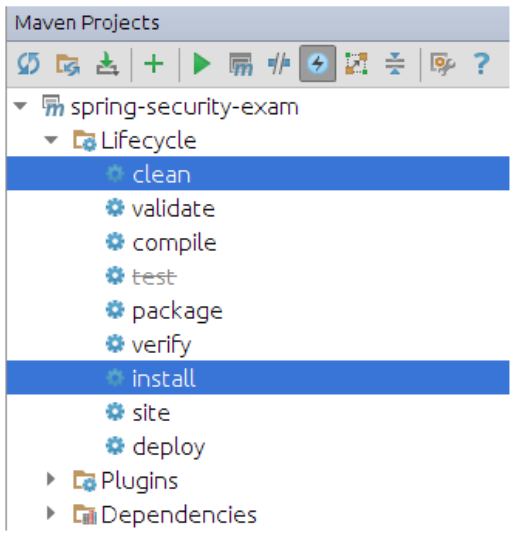
```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

```
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

</web-app>
```

Шаг 11. Развертывание и тестирование

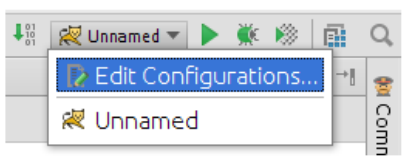
Пора все собрать и развернуть на сервере приложений.

Собираем проект с помощью Maven:

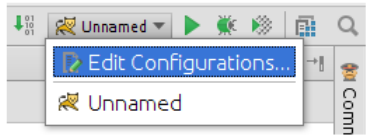


Теперь развертываем собранный собраный war на сервере приложений, я выбрал Tomcat 8.

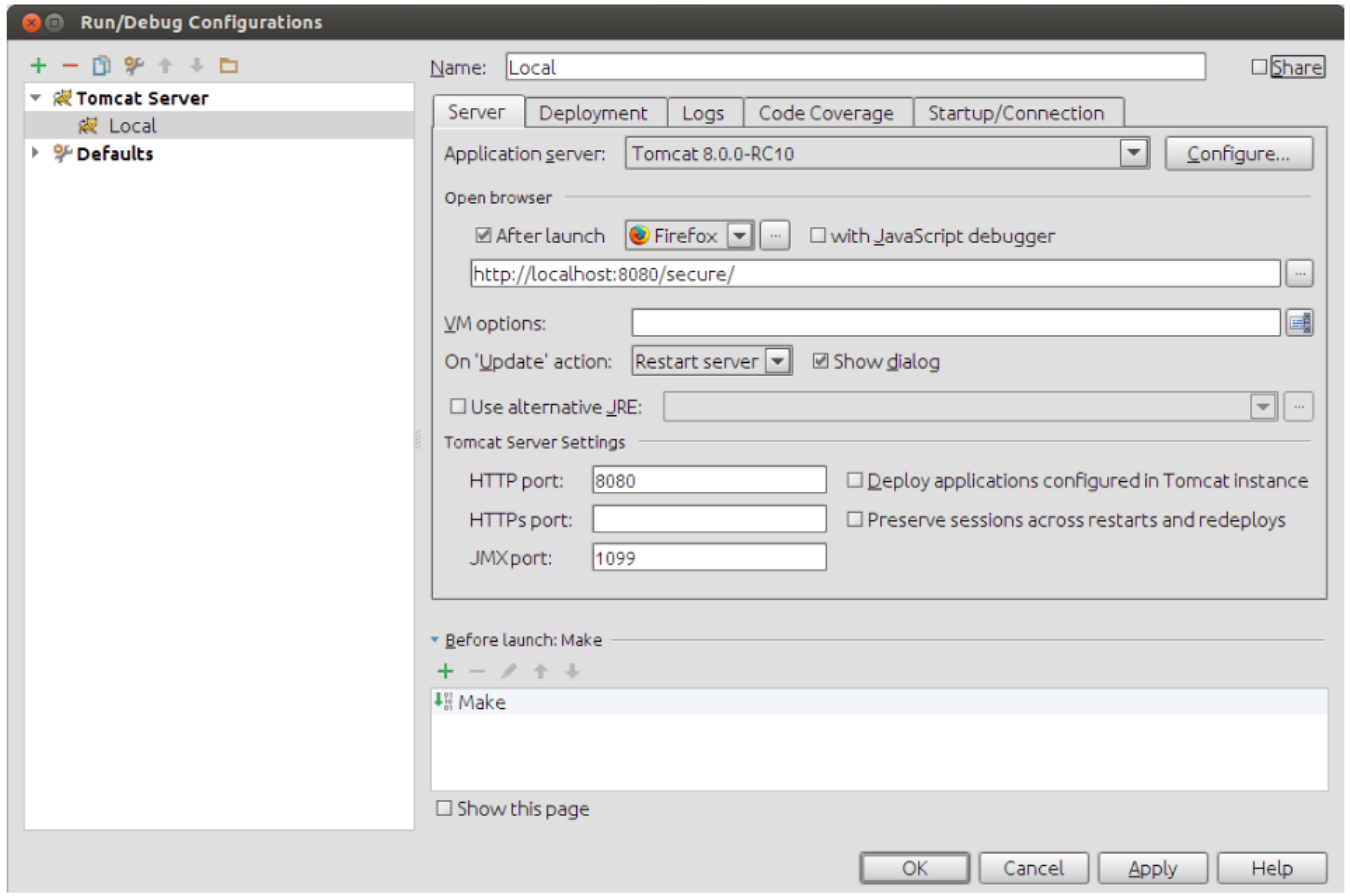
Конфигурируем Intellij IDEA под Tomcat. Заходим в **Edit Configuration**...



Конфигурируем Intellij IDEA под Tomcat. Заходим в **Edit Configuration...**

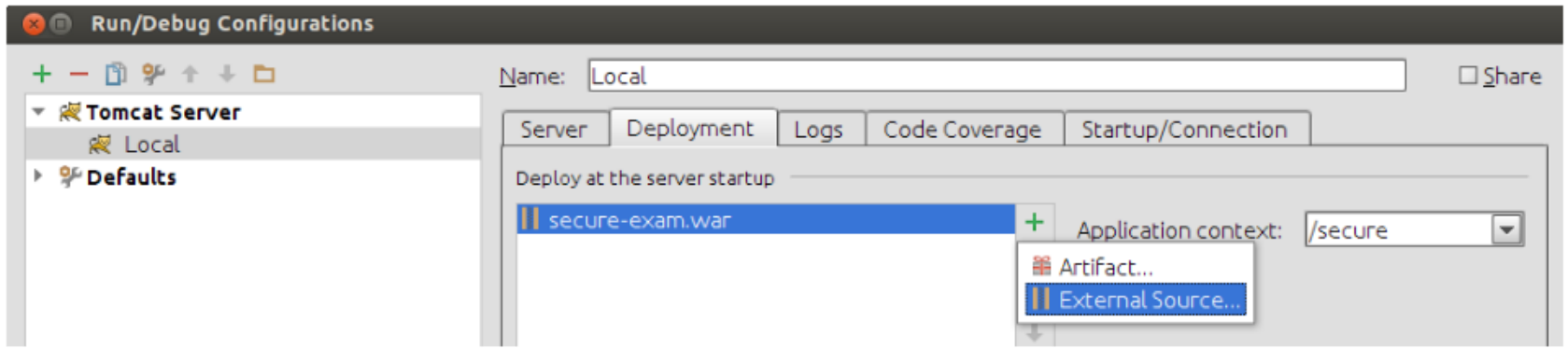


Нажимаем по плюсику (**Add New Configuration...**) ищем там **Tomcat Server -> Local:**



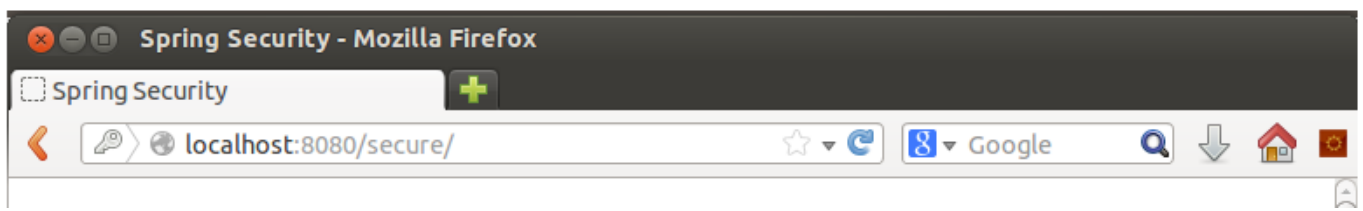
Конфигурируем также как показанно на скриншоте, потом переходим в раздел **Deployment**:

Конфигурируем также как показанно на скриншоте, потом переходим в раздел **Deployment**:

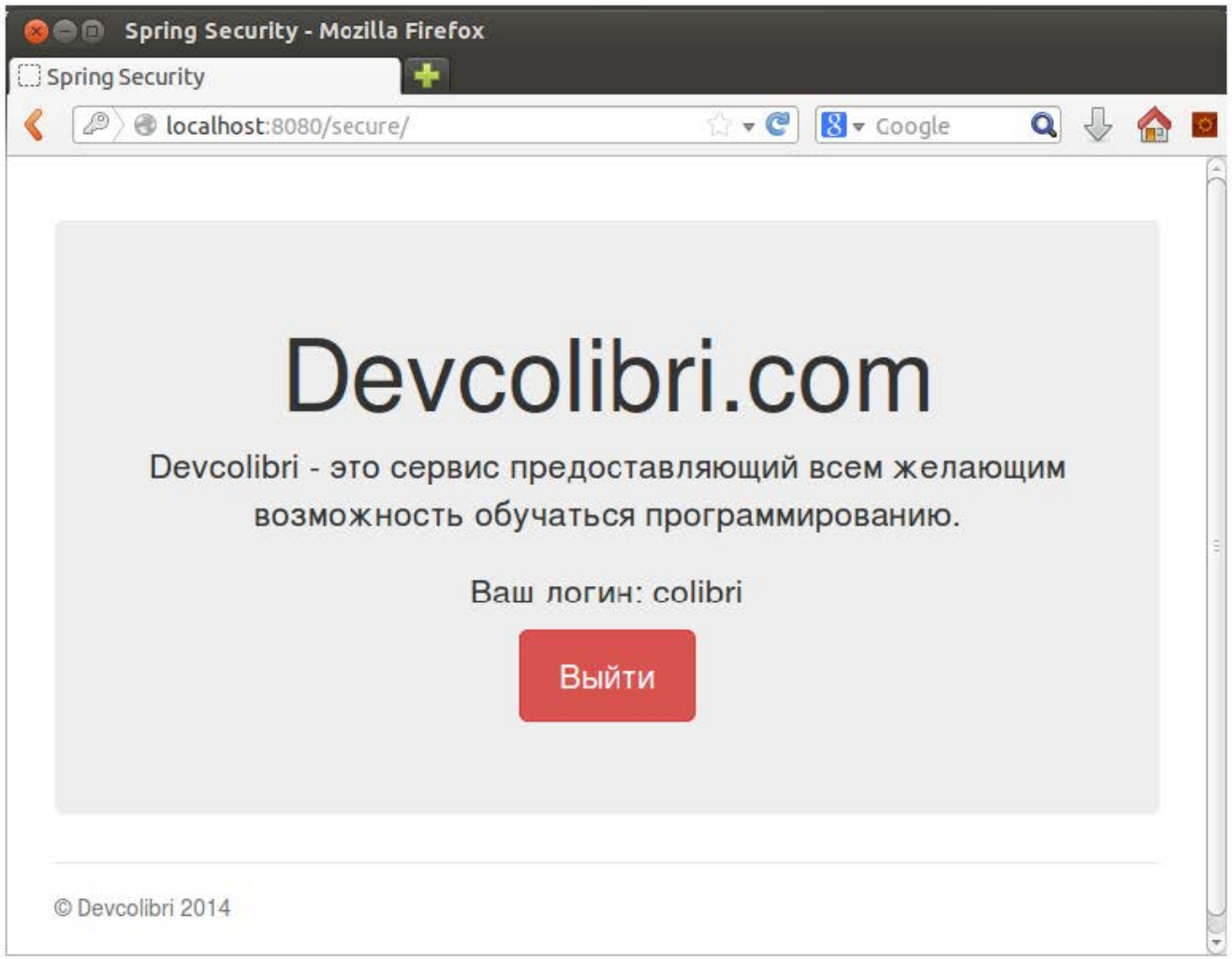


Жмем **Add -> External Source...** в корне вашего проекта будет папка **target** в ней будет **secure-exam.war** выбираем его и в поле **Application context** пишем **/secure.**

После этого запускаем. Попадпем на главное окно жмем Войти переходим на форму с логином и нажимаем Войти, после этого вас перенаправит на главную страницу где вы увидите свой логин.



После этого запускаем. Попадпем на главное окно жмем Войти переходим на форму с логином и нажимаем Войти, после этого вас перенаправит на главную страницу где вы увидите свой логин.



На этом все.