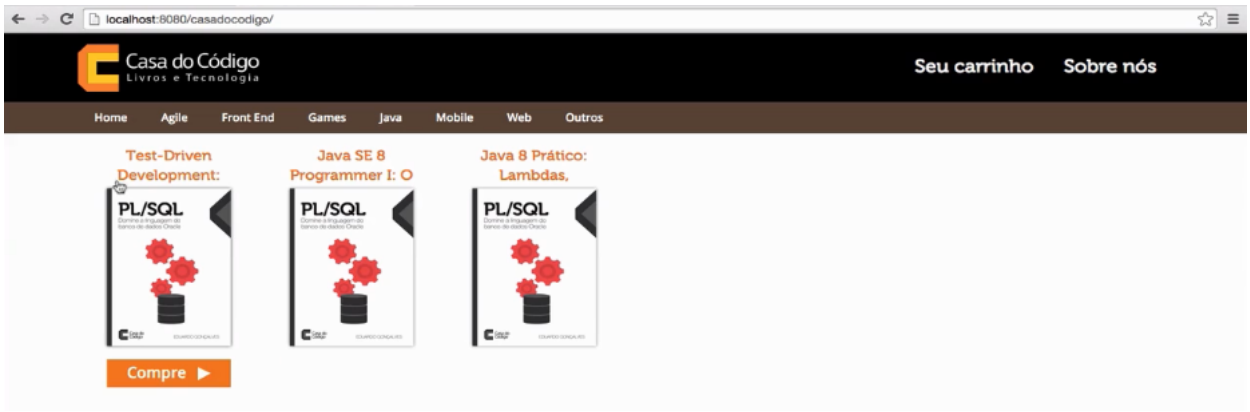
07

Personalizando o login e logout

Personalizando Login e Logout

Já temos a listagem e cadastro de produtos funcionando perfeitamente, mas note que não temos os links de acesso a estas páginas, na Home do projeto. Atualmente, a página inicial (`home.jsp`) encontra-se dessa forma:



Vamos adicionar os links para as páginas de cadastro e listagem de produtos junto aos links de `Seu carrinho` e `Sobre nós` . No arquivo `home.jsp` , busque o seguinte trecho de código:

```
<ul class="clearfix">
    <li><a href="/cart" rel="nofollow">Carrinho</a></li>
    <li><a href="/pages/sobre-a-casa-do-codigo" rel="nofollow">Sobre Nós</a></li>
</ul>
```

Vamos criar mais duas opções neste menu com os links que criamos na aula passada apontando para as páginas de listagem e cadastro de produtos da seguinte forma:

```
<ul class="clearfix">

    <li><a href="${s:mvcUrl('PC#listar').build() }" rel="nofollow">Listagem de Produtos</a></li>
    <li><a href="${s:mvcUrl('PC#form').build() }" rel="nofollow">Cadastro de Produtos</a></li>

    <li><a href="/cart" rel="nofollow">Carrinho</a></li>
    <li><a href="/pages/sobre-a-casa-do-codigo" rel="nofollow">Sobre Nós</a></li>
</ul>
```

E como resultado teremos os links sendo exibidos na página inicial da aplicação:

Vamos refletir sobre o que acabamos de fazer: com os links expostos desta maneira, possibilitamos que qualquer usuário possa cadastrar livros na Casa do Código. Imagine a Casa do Código recebendo pedidos de livros que não existem realmente. Precisamos ter um controle de acesso e somente pessoas autorizadas poderão entrar nas páginas de cadastro de livros, assim como na de listagem.

Há várias estratégias que resolvem este tipo de problema. Poderíamos criar um filtro de requisições, um interceptador etc. Mas usaremos um recurso que o próprio *Spring* nos fornece. Um filtro já pronto, capaz de fazer o trabalho de verificar se o usuário tem ou não autorização de acessar determinadas páginas.

Para utilizarmos este filtro, precisamos configura-lo no projeto de forma semelhante ao que fizemos quando criamos a classe `ServletSpringMVC` , ou seja, criar uma outra classe que herde da classe do filtro que já se encontra configurado.

Para que sejamos capazes de utilizar os recursos de segurança do *Spring*, teremos que deixar estes recursos declarados nas depêndencias do nosso projeto. Sendo assim, devemos abrir o arquivo `pom.xml` e adicionar as seguintes linhas de dependência:

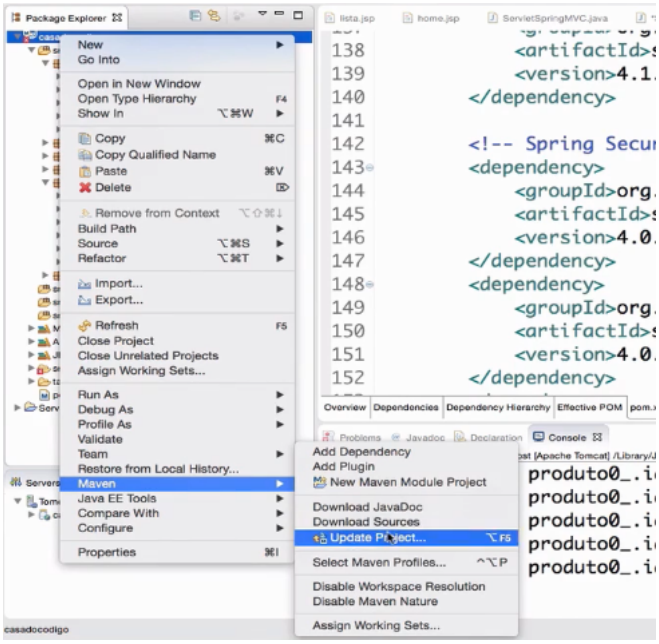
```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>4.0.0.M2</version>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-taglibs</artifactId>
    <version>4.0.0.M2</version>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>4.1.1.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
    <version>4.0.0.M2</version>
</dependency>
```

E fora da tag `dependencies` devemos adicionar o repositório de onde o **Maven** irá baixar estas bibliotecas.

```
<repositories>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone/</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
```

Observação: Note que a biblioteca usada é da versão **Milestone**, por ser a que estava disponível durante a criação do curso. Caso encontre a biblioteca em sua versão final, use-a, a mesma deve funcionar sem problemas. Caso tenha problemas, use nosso forúm e tire suas dúvidas!

Após as adições no arquivo `pom.xml` lembre-se de atualizar o projeto da seguinte forma:



O próximo passo para começarmos a utilizar realmente os recursos de segurança do *Spring* é criar a classe responsável por inicializar o filtro de segurança. No pacote `br.com.casadocodigo.loja.conf`, criaremos a classe `SpringSecurityFilterConfiguration`, que faremos extender a classe `AbstractSecurityWebApplicationInitializer`.

```
public class SpringSecurityFilterConfiguration extends AbstractSecurityWebApplicationInitializer{

}
```

Apenas isto não será o suficiente. A classe da forma que está já funciona, mas ela apenas inicializa o filtro de segurança do *Spring*. Onde estão realmente as configurações de segurança? Em nenhum lugar! Para armazenar as configurações de segurança, criaremos uma nova classe chamada `SecurityConfiguration` no mesmo pacote, iremos anotá-la com `@EnableWebMvcSecurity`.

```
@EnableWebMvcSecurity
public class SecurityConfiguration {

}
```

Desta forma, o *Spring* através da classe com esta anotação já configura alguns detalhes de segurança de forma automática. Mas para que isso funcione, o *Spring* precisa saber que a classe existe. Lembra qual classe que carrega todas as configurações de nossa aplicação? É a `ServletSpringMVC` . Nesta usamos o método `getServletConfigClasses` para carregar as configurações da aplicação e do **JPA** no primeiro módulo deste curso.

Mas agora, o método que usaremos é o `getRootConfigClasses` que carrega configurações logo ao iniciar a aplicação. Este método simplesmente retorna um **Array** de classes do mesmo jeito que o método `getServletConfigClasses` faz. Assim faremos a classe `SecurityConfiguration` ser reconhecida pelo *Spring*:

```
protected Class<?>[] getRootConfigClasses() {
    return new Class[]{SecurityConfiguration.class};
}
```

Agora poderemos fazer o teste! Mas ao reiniciarmos o servidor, teremos um erro indicando que nossas configurações não foram suficientes, veja:

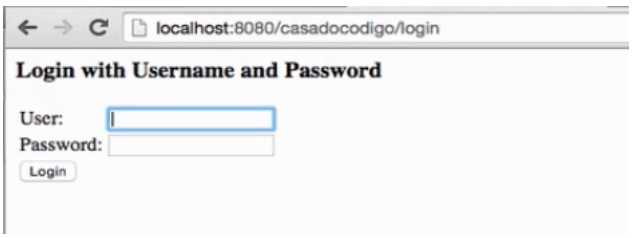
O erro explica que deve haver uma classe que herde da classe `WebsecurityConfigurerAdapter` . Se olharmos nossa classe de configuração (`SecurityConfiguration`), veremos que ela só habilita as configurações, mas não configura nada. Veja:

```
@EnableWebMvcSecurity
public class SecurityConfiguration {

}
```

A anotação é bem clara. Ela simplesmente habilita o recurso de **Web MVC Secutiry** e quem configura o recurso é justamente a classe `WebSecurityConfigurerAdapter` . Então, para solucionar o erro, precisaremos fazer com que a classe `SecurityConfiguration` herde da classe `WebSecurityConfigurerAdapter` da seguinte maneira:

```
@EnableWebMvcSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter{

}
```

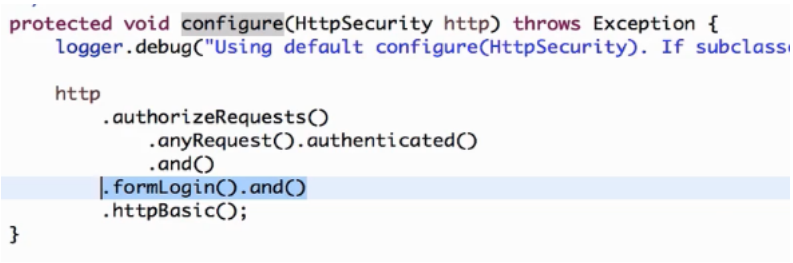
Se reiniciarmos o servidor agora, já poderemos ver que o erro foi resolvido e que o recurso de segurança já funciona. Ao abrirmos nossa aplicação em `localhost:8080/casadocodigo` teremos:



Agora ao tentarmos acessar qualquer página no projeto, seremos redirecionados para esta tela de login e para realizar qualquer operação, teremos que nos autenticar. Isso não é interessante! Nós queremos que os usuários acessem os livros e o

detalhes dos mesmos, o carrinho de compras e outras páginas sem impedimentos. Vamos voltar para o código e ver como podemos fazer este comportamento funcionar.

A classe `SecurityConfiguration` herda um método chamado `configure` que em sua implementação padrão, bloqueia e redireciona todas as requisições não autenticadas. Veja o código deste método abaixo:



O que podemos fazer é sobrescrever este método em nossa classe de configuração e descrever os padrões de **URLs** que queremos ou não bloquear. Estes padrões são definididos através do método `antMatchers` da classe `HttpSecurity`. Veja o código abaixo:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/produtos/form").hasRole("ADMIN")
    .antMatchers("/carrinho/**").permitAll()
    .antMatchers(HttpMethod.POST, "/produtos").hasRole("ADMIN")
    .antMatchers(HttpMethod.GET, "/produtos").hasRole("ADMIN")
    .antMatchers("/produtos/**").permitAll()
    .antMatchers("/").permitAll()
    .anyRequest().authenticated()
    .and().formLogin();
}
```

Perceba que montamos uma série de padrões de caminhos definindo onde será permitido o acesso com autenticação ou não. Estamos bloqueando o acesso a `/produtos/form` para todos que não são **ADMIN**, assim como requisições do tipo **POST** para o caminho `/produtos/form`. Estamos permitindo o acesso ao **home** da nossa aplicação através do `/` e também ao **carrinho de compras** através do `/carrinho/**`. Por último, estamos sinalizando que estas verificações devem ser feitas para todas as requisições e que as bloqueadas através do `hasRole` devem ser autenticadas.

A cada requisição, a verificação é feita e caso uma das páginas bloqueadas tenha tentativas de acesso sem autenticação, estas serão redirecionadas para o formulário de login. Experimente testar quais páginas estão ou não bloqueadas agora. Lembre-se de reiniciar o servidor.

Se verificarmos a página inicial novamente, veremos que os links para a listagem e cadastro de produtos continuam a ser exibidos apesar de sermos direcionados para o formulário de login ao clicá-los.

Vamos refletir sobre o **login**: ele precisa de usuários que possam se autenticar, caso contrário, os links administrativos da aplicação ficariam inacessíveis. Ainda não temos usuários no banco de dados - nem mesmo temos classes que representem usuários em nosso projeto. Falta criar estas classes!

Criaremos primeiramente a classe `UsuarioDAO`, responsável por manipular os dados dos usuários no banco de dados. Criaremos esta classe no pacote `br.com.casadocodigo.loja.daos` e a anotaremos com `@Repository`. Definiremos também um atributo privado do tipo `EntityManager` que chamaremos de `manager` e o anotaremos com `PersistenceContext`.

```
@Repository
public class UsuarioDAO {

    @PersistenceContext
    private EntityManager manager;

}
```

Em seguida, criaremos o método `find` que recebe um parâmetro do tipo `String` que será o `email` do usuário a ser buscado no banco de dados. Este método deve retornar um objeto do tipo `Usuario` e usar o objeto `manager` para criar uma consulta (`createQuery`) que busque os usuários no banco de dados e os armazene em uma lista de usuários.

```
public Usuario find(String email){
        List<Usuario> usuarios = manager.createQuery("select u from Usuario u where u.email = :emai
                .setParameter("email", email)
                .getResultList();
}
```

Após receber o resultado da consulta ao banco de dados, precisaremos verificar se algum usuário foi encontrado, isto é, verificar se a lista está vazia (`isEmpty`) e caso esteja, lançaremos uma exceção informando que o usuário com o email enviado não foi encontrado. Caso a lista não esteja vazia, retornaremos o primeiro item da lista.

```
public Usuario find(String email){
    List<Usuario> usuarios = manager.createQuery("select u from Usuario u where u.email = :email", l
            .setParameter("email", email)
            .getResultList();

    if(usuarios.isEmpty()){
        throw new RuntimeException("O usuário "+ email +" não foi encontrado");
    }

    return usuarios.get(0);
```

```
}
```

Note que o *Eclipse* a todo momento reclama do nosso código, marcando de vermelho algumas partes. Isto acontece, porque nesta nova classe estamos referenciando uma outra classe que ainda não existe - a classe `Usuario`, que criaremos adiante.

A classe usuário por hora apenas terá alguns atributos e será anotada com `@Entity`, estes atributos serão o `email`, `senha`, `nome`, e uma lista que chamaremos de `roles` que guardará objetos do tipo `Role`. Usaremos o `email` como identificador único para cada usuário, sendo assim, o marcaremos com a anotação `@Id`. Assim teremos:

```
@Entity
public class Usuario {
    @Id
    private String email;
    private String nome;
    private String senha;
    private List<Role> roles = new ArrayList<Role>();

}
```

Lembre-se que a classe usuário deve ser criada no pacote `br.com.casadocodigo.loja.models`. Criaremos também a classe `Role` que representará permissões do usuário. Esta classe apenas terá um atributo do tipo `String` que será o `nome` da permissão que será anotada com `@Id`. A classe também será anotada com `@Entity` e ficará no mesmo pacote da classe `Usuario`.

```
@Entity
public class Role {
    @Id
    private String nome;
}
```

Com os atributos das classes `Usuario` e `Role` privados, não podemos acessá-los, por isso devemos gerar os *Getters and Setters* para estas classes. Por fim teremos:

```
@Entity
public class Usuario {
    @Id
    private String email;
    private String nome;
    private String senha;
    private List<Role> roles = new ArrayList<Role>();

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getNome() {
        return nome;
```

```
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getSenha() {
        return senha;
    }

    public void setSenha(String senha) {
        this.senha = senha;
    }

    public List<Role> getRoles() {
        return roles;
    }

    public void setRoles(List<Role> roles) {
        this.roles = roles;
    }

}


@Entity
public class Role {

    @Id
    private String nome;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

}
```

As regras para um usuário se autenticar em nossa aplicação já estão prontas, porém, precisamos fazer algumas configurações para que o *Spring* consiga utilizar a classe `UsuarioDAO`. Esta configuração se dá pelo uso do `UserDetailsService` que é uma interface do *Spring* que realmente trabalha com as configurações de autenticação.

O primeiro passo é sobrescrever o método `configure` na classe `SecurityConfiguration`, que recebe um objeto do tipo `AuthenticationManagerBuilder` chamado de `auth` e usar o método `userDetailsService` passando para este método um objeto do tipo `UsuarioDAO`. Criaremos o objeto como um atributo da classe `SecurityConfiguration` e o anotaremos com `@Autowired`.

```
@EnableWebMvcSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter{

    @Autowired
    private UsuarioDAO usuarioDao;
```

```
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        [...]
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(usuarioDao);
    }
}
```

Isso apenas não é suficiente para configurar a autenticação dos usuários. Aconteque que o método `userDetailsService` espera receber um objeto que implemente uma interface com este mesmo nome. Faremos então classe `UsuarioDAO` implementar a interface e adicionar à classe os método que precisam ser implementados.

```
public class UsuarioDAO implements UserDetailsService {

    @PersistenceContext
    private EntityManager manager;

    public Usuario find(String email){
        [...]
    }

    @Override
    public UserDetails loadUserByUsername(String arg0) throws UsernameNotFoundException {
        // TODO Auto-generated method stub
        return null;
    }

}
```

Para que não implementemos todo o método `loadUserByUsername` , simplesmente iremos trocar o nome do método `find` para `loadUserByUsername` e trocar o lançamento do `RuntimeException` para o `UsernameNotFoundException` com a mesma mensagem. Assim teremos:

```
@Repository
public class UsuarioDAO implements UserDetailsService {

    @PersistenceContext
    private EntityManager manager;

    public UserDetails loadUserByUsername(String email) {
        List<Usuario> usuarios = manager.createQuery("select u from Usuario u where u.email = :email
                .setParameter("email", email).getResultList();

        if (usuarios.isEmpty()) {
            throw new UsernameNotFoundException("O usuário " + email + " não foi encontrado");
        }

        return usuarios.get(0);
    }
```

```
}
```

É importante notar que o retorno do método também mudou. O mesmo não retorna mais um objeto do tipo `Usuario`, mas sim do tipo `UserDetails`. Sendo assim, teremos que fazer com que a classe `Usuario` implemente esta interface e adicione seus métodos.

```
@Entity
public class Usuario implements UserDetails {
    @Id
    private String email;
    private String nome;
    private String senha;
    private List<Role> permissoes = new ArrayList<Role>();

    // getters and setters

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public String getPassword() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public String getUsername() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public boolean isAccountNonExpired() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean isAccountNonLocked() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean isEnabled() {
        // TODO Auto-generated method stub
```

```
            return false;
        }

    }
```

Para os métodos `isAccountNonExpired`, `isAccountNonLocked`, `isCredentialsNonExpired` e `isEnabled` retornaremos `true`, por não termos nenhum controle para expiração, bloqueio de contas ou expiração de credenciais dos usuários e queremos que os mesmos sempre esteja ativos.

Para os métodos `getUsername`, `getPassword` e `getAuthorities` retornaremos respectivamente o `email`, `senha` e `roles`. Com a série de mudanças, a classe `Usuario` ficará da seguinte forma:

```
@Entity
public class Usuario implements UserDetails {
    @Id
    private String email;
    private String nome;
    private String senha;
    private List<Role> roles = new ArrayList<Role>();

    // getters and setters

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return this.roles;
    }

    @Override
    public String getPassword() {
        return this.senha;
    }

    @Override
    public String getUsername() {
        return this.email;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
```

```
    }

}
```

Por algum motivo o método `getAuthorities` está com erros. Se analisarmos a assinatura deste método, veremos que este retorna uma coleção de objetos do tipo `GrantedAuthority`. Para resolvermos o problema, basta fazer com que a classe `Role` implemente a interface e adicione seus métodos. Esta interface tem apenas um método chamado `getAuthority`, no qual retornaremos o atributo `nome` da classe `Role`.

```
@Entity
public class Role implements GrantedAuthority{

    @Id
    private String nome;

    // getters and setters

    @Override
    public String getAuthority() {
        return this.nome;
    }

}
```

O *Eclipse* agora marca as classes `Usuario` e `Role` de amarelo recomendando que estas tenham um atributo do tipo `SerialVersionId`. Adicionaremos os mesmos com a ajuda do *Eclipse*. Assim teremos:

```
@Entity
public class Role implements GrantedAuthority{
    private static final long serialVersionUID = 1L;
    [...]
}
```

E na classe `usuario`:

```
@Entity
public class Usuario implements UserDetails {
    private static final long serialVersionUID = 1L;
    [...]
}
```

E por último passo, precisaremos da configuração de codificação das senhas dos usuários. Usaremos uma criptografia chamada **BCrypt** e faremos isto utilizando a classe `BCryptPasswordEncoder`. Em seguida, vamos usar o método `passwordEncoder` do objeto `auth`, no método `configure` da classe `SecurityConfiguration`.

```
@EnableWebMvcSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter{

    @Autowired
    private UsuarioDAO usuarioDao;
```

```
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        [...]
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(usuarioDao)
        .passwordEncoder(new BCryptPasswordEncoder());
    }
}
```

Uma observação deve ser feita antes que tentemos testar as funcionalidades de autenticação de usuários: as configurações de segurança estão iniciando no método `getRootConfigClasses` da classe `ServletSpringMVC` e estas requerem as configurações do *DAO*, carregadas em outro momento pelo método `getServletConfigClasses`. Isto deve gerar erros ao tentarmos inicializar a aplicação. Para resolvermos,colocaremos todas as configurações para inicializar no método `getRootConfigClasses`. A classe `ServletSpringMVC` deve ficar da seguinte forma:

```
public class ServletSpringMVC extends AbstractAnnotationConfigDispatcherServletInitializer{

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{SecurityConfiguration.class, AppWebConfiguration.class, JPAConfiguration
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] {};
    }
    [...]
}
```

Temos ainda um segundo problema: nós não fizemos o mapeamento das classes `Usuario` com a classe `Role`. O mapeamento será de um usuário para muitas permissões, no caso `roles`. Queremos também que ao carregar o usuário, as permissões sejam carregadas por meio do `fetch=FetchType.EAGER`. A classe `Usuario` ficará assim:

```
@Entity
public class Usuario implements UserDetails {
    private static final long serialVersionUID = 1L;

    @Id
    private String email;
    private String nome;
    private String senha;

    @OneToMany(fetch=FetchType.EAGER)
    private List<Role> roles = new ArrayList<Role>();

    [...]
}
```

Agora sim! Após estes ajustes, podemos testar nossa aplicação sem esperar por erros, considerando que nos adiantamos sobre alguns pontos. Ao iniciarmos a aplicação, veremos que tudo funciona perfeitamente, as páginas que bloqueamos requerem autenticação e as outras estão exibindo as informações de forma normal. Mas podemos testar se o login está realmente funcionando? Perceba que não temos usuários cadastrados no banco de dados. Precisaremos fazer isso manualmente.

Vamos abrir o console, selecionar o banco de dados e executar as seguintes consultas.

Cadastrar um *Role* de **ADMIN**.

```
insert into Role values ('ROLE_ADMIN');
```

Note que o nome do *Role* segue um padrão: **ROLE_ALGUMACOISA**. Isso porque o método `hasRole` que usamos nas verificações segue o mesmo padrão. Agora precisamos cadastrar um usuário.

```
insert into Usuario (email, nome, senha) values ('admin@casadocodigo.com.br', 'Administrador', '$2a
```

Aqui o único ponto que merece uma observação é a senha. Esse código esquisito nada mais é o resultado da criptografia BCrypt dos carecteres **123456**. Pode-se gerar esses códigos com algumas ferramentas ou linguagens de programação. uma das ferramentas é o **BCrypt Calculator** (https://www.dailycred.com/article/bcrypt-calculator).

Por último, precisamos relacionar o usuário com a *role* da seguinte forma:

```
insert into Usuario_Role(Usuario_email, roles_nome) values ('admin@casadocodigo.com.br', 'ROLE_ADMI
```

Já podemos testar, sem precisar reiniciar o servidor, pois a modificação foi simplesmente no banco de dados. Agora ao sermos redirecionados para o formulário de Login, podemos usar como usuário o email: **admin@casadocodigo.com.br** e como senha: **123456**. Após autenticar o usuário seremos redirecionados para a página que antes estava bloqueada.

Restringindo Conteúdo

Na página inicial da nossa aplicação, os links para listagem e cadastro de produtos ainda é visível para quem não está logado, ou seja, estão públicos e isso não é bom. Devemos esconde-los e somente exibi-los se o usuário estiver logado.

O *Spring* nos fornece uma *Taglib* que faz exatamente o que queremos fazer. Esta que pode ser importada na página `home.jsp` da seguinte maneira:

```
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="security" %>
```

Ao importar a *Taglib*, podemos fazer uso da tag `<security:authorize>` que tem o atributo `access` no qual podemos usar o método `isAuthenticated` também fornecido pela própria *Tablib* do *Spring*. Assim devemos encontrar o seguinte trecho de código:

```
<div id="header-content">
    <nav id="main-nav">
        <ul class="clearfix">
            <li><a href="${s:mvcUrl('PC#listar').build() }" rel="nofollow">Listagem de Produtos</a>
            <li><a href="${s:mvcUrl('PC#form').build() }" rel="nofollow">Cadastro de Produtos</a></l
            <li><a href="/cart" rel="nofollow">Carrinho</a></li>
            <li><a href="/pages/sobre-a-casa-do-codigo" rel="nofollow">Sobre Nós</a></li>
        </ul>
    </nav>
</div>
```

E esconder os links de listagem e cadastro de produtos da seguinte forma:

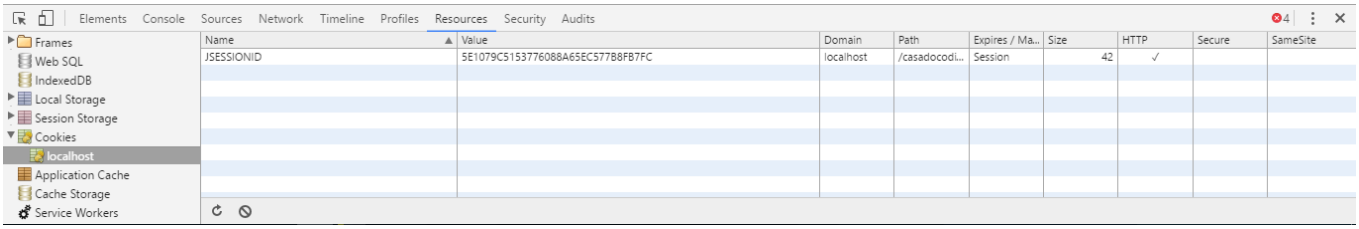
```
<security:authorize access="isAuthenticated()">
    <li><a href="${s:mvcUrl('PC#listar').build() }" rel="nofollow">Listagem de Produtos</a></li>
    <li><a href="${s:mvcUrl('PC#form').build() }" rel="nofollow">Cadastro de Produtos</a></li>
</security:authorize>
```

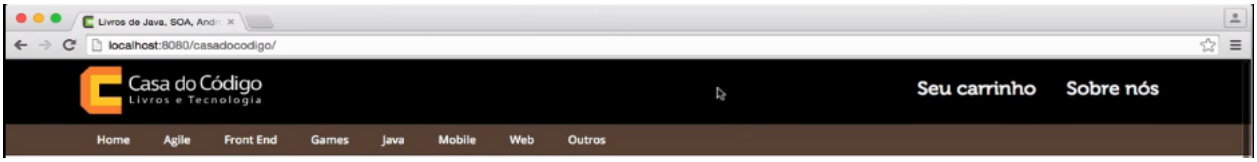
O menu completo da página `home.jsp` ficará da seguinte forma:

```
<div id="header-content">
    <nav id="main-nav">
        <ul class="clearfix">
            <security:authorize access="isAuthenticated()">
                <li><a href="${s:mvcUrl('PC#listar').build() }" rel="nofollow">Listagem de Produtos
                <li><a href="${s:mvcUrl('PC#form').build() }" rel="nofollow">Cadastro de Produtos</a
            </security:authorize>
            <li><a href="/cart" rel="nofollow">Carrinho</a></li>
            <li><a href="/pages/sobre-a-casa-do-codigo" rel="nofollow">Sobre Nós</a></li>
        </ul>
    </nav>
</div>
```

Mas ao atualizarmos a página, não teremos nenhum resultado visual já que estamos logados e não implementamos nenhuma forma de deslogar da aplicação ainda. Como faremos para verificar se os links realmente não aparecem para os usuários não logados? A forma mais simples é apagando o **Cookie** de sessão do *Spring*.

Abrindo as ferramentas do desenvolvedor do navegador, podemos navegar até a sessão `resourses -> cookies -> localhost` e apagar o *Cookie* com o nome `JSESSIONID`. Assim, ao atualizarmos a página não veremos mais os links que ocultamos anteriormente.





Mostrando dados do usuário logado.

Uma caracteristica interessante para adicionarmos na nossa aplicação será mostrar qual usuário está logado no momento. O lugar em que o nome do usuário costuma ser exibido nas aplicações é dentro do painel de administração. No caso, pode ser a página de listagem dos produtos, no canto superior, do lado direito do menu principal.

Para exibirmos dados do usuário logado, as tags de segurança do *Spring* nos ajudam de uma forma bastante simples. Existe uma tag chamada `security:authentication`, na qual através do atributo `property` podemos usar o valor `principal` para recuperar dados do usuário atualmente logado.

Vamos abrir a página `lista.jsp` e encontrar o seguinte treco de código:

```
<div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
  <ul class="nav navbar-nav">
    <li><a href="${s:mvcUrl('PC#listar').build() }">Lista de Produtos</a></li>
    <li><a href="${s:mvcUrl('PC#form').build() }">Cadastro de Produtos</a></li>
  </ul>
</div>
```

Logo abaixo da tag `ul` criaremos uma nova `ul` com as classes padrões do bootstrap para criação de um menu alinhado à direita da barra de menu.

```
<ul class="nav navbar-nav navbar-right">
  <li><a href="#">NOME DO USUÁRIO</li>
</ul>
```

Neste novo trecho de código, substituiremos `NOME DO USUÁRIO` pela tag `security:authentication`. Usaremos o atributo `property` e através do valor `principal`, vamos recuperar o `username` atual do usuário, da seguinte forma:

```
<ul class="nav navbar-nav navbar-right">
  <li><a href="#"><security:authentication property="principal.username"/></a></li>
</ul>
```

Visualmente, já podereamos verificar o resultado após logar na aplicação.

Uma variação do código anterior é usar outra variável para guardar os dados do usuário. Podemos fazer isto da seguinte forma para alcançar o mesmo resultado:

```
<ul class="nav navbar-nav navbar-right">
  <li>
    <a href="#">
        <security:authentication property="principal" var="usuario"/>
        Usuário: ${usuario.username}
    </a>
  </li>
</ul>
```

Outra variação de código que pode ser usada para exibir (ou não) informações nas páginas `jsps` é o uso do `hasRole` na tag `security:authorize`, passando a *role* requerida para o acesso. No caso de ocultar os links da página `home.jsp`, como fizemos antes, podemos fazer dessa maneira:

```
<security:authorize access="hasRole('ROLE_ADMIN')">
    <li><a href="${s:mvcUrl('PC#listar').build() }" rel="nofollow">Listagem de Produtos</a></li>
    <li><a href="${s:mvcUrl('PC#form').build() }" rel="nofollow">Cadastro de Produtos</a></li>
</security:authorize>
```

Neste caso o uso do paramêtro para o `hasRole` precisa ser prefixado com o **ROLE_**.

Cross Site Request Forgery

A aplicação aparenta estar funcionando normalmente, mas o que acontece se tentarmos realizar uma compra ou tentarmos pelo menos adicionar um produto ao carrinho?

O que antes funcionava, agora não funciona. Adicionar os componentes de segurança do *Spring* fez com que o formulário de adição de produtos no carrinho parasse de funcionar por algum motivo. O que será esse **CSRF**?

CSRF é uma sigla que representa a frase **Cross Site Request Forgery**, uma técnica de ataque a sites muito comum na web. A técnica representa o cenário de que um outro site está enviando dados para nossa aplicação, em vez do usuário diretamente. Geralmente, acontece com páginas clonadas, uma página falsa é apresentada ao usuário e este sem saber submete seus dados que por sua vez podem ser enviados ao servidor original ou podem ser feitas cópias em um servidor falso para posteriormente ter o uso indevido.

Por ser uma técnica de ataque muito comum e também muito perigosa, diversos frameworks de várias plataformas, já possume recursos prontos para que usemos afim de evitar tais ocorrências. Com o *Spring* não é diferente.

A solução mais simples é criar um novo `input` no formulário com o valor `${_csrf.parameterName}` no atributo `name` e `${_csrf.token}`. Em nossa página de `detalhes.jsp` teriamos:

```
<form action="<c:url value='/carrinho/add' />" method="post" class="container">
    <ul id="variants" class="clearfix">
        <input type="hidden" name="produtoId" value="${produto.id}" />
        <c:forEach items="${produto.precos}" var="preco">
          <li class="buy-option">
           [...]
          </li>
        </c:forEach>
    </ul>
    <button type="submit" class="submit-image icon-basket-alt" alt="Compre Agora" title="Compre Agor

    <input type="hidden" name="${_csrf.parameterName }" value="${_csrf.token }" />

  </form>
```

Com este código, o *Spring* irá gerar o `name` e o `value` de forma que ele saiba se o **POST** veio do formulário da página ou de algum outro lugar. O `_csrf.token` nada mais gera do que uma sequência de números verificadores que se alterados tornam-se inválidos gerando o seguinte erro:

Mas convenhamos de que podemos facilmente esquecer de criar o `input` de *token* manualmente toda vez que precisarmos criar um formulário em nossa aplicação. Para que evitemos esses casos de esquecimento, podemos usar uma segunda forma de criação de formulários que já vem com este campo configurado. Usando as tags de formulários do próprio *Spring*. Assim só precisamos alterar a tag `form` do formulário de compras da página `detalhes.jsp`.

```
<form:form servletRelativeAction="/carrinho/add" method="post" cssClass="container">
    <ul id="variants" class="clearfix">
        <input type="hidden" name="produtoId" value="${produto.id}" />
        <c:forEach items="${produto.precos}" var="preco">
          [...]
        </c:forEach>
    </ul>
    <button type="submit" class="submit-image icon-basket-alt" alt="Compre Agora" title="Compre Agor
  </form:form>
```

Dessa forma não precisamos nos preocupar em criar um campo *CSRF* manualmente toda vez que formos adicionar um formulário em alguma página, o *Spring* já fará isso automaticamente. Perceba que utilizamos uma forma alternativa de criar o `action` do formulário através do `servletRelativeAction` ao invés do `mvcUrl`.

Personalizando login e logout

Não foi comentado anteriormente mas é curioso. De onde veio o formulário de login que utilizamos para testar e autenticar usuários em nossa aplicação? Do próprio *Spring*! Este que apesar de funcionar, não é o ideal. Queremos nosso próprio formulário com os estilos da casa do código. Vamos criar então uma nova página chamada `loginForm` dentro da pasta `webapp/WEB_INF/views` e dentro deste novo *JSP* usaremos o seguinte código:

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


<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Livros de Java, Android, iPhone, Ruby, PHP e muito mais - Casa do Código</title>
    <c:url value="/resources/css" var="cssPath" />
    <link rel="stylesheet" href="${cssPath}/bootstrap.min.css" />
    <link rel="stylesheet" href="${cssPath}/bootstrap-theme.min.css" />
    <style type="text/css">
        body{
            padding: 60px 0px;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>Login Casa do Código</h1>
        <form:form servletRelativeAction="/login" method="post">
            <div class="form-group">
                <label>Nome</label>
                <input type="text" name="username" class="form-control" />
            </div>
            <div class="form-group">
                <label>Senha</label>
                <input type="password" name="password" class="form-control" />
            </div>
            <button type="submit" class="btn btn-primary">Logar</button>
        </form:form>
    </div>
</body>
</html>
```

E para que possamos utilizar essa nova **View**, devemos criar um novo **Controller** que chamaremos de `LoginController` que será anotado com `@Controller` e terá o método `loginForm` que retorna apenas o nome da *view* do login e será mapeado para `/login` aceitando apenas requisições do tipo **GET**.

```
@Controller
public class LoginController {

    @RequestMapping(value="/login", method=RequestMethod.GET)
    public String loginForm(){
        return "loginForm";
    }

}
```

Fazer isso ainda não é o suficiente, precisamos de alguma forma informar o *Spring* de que ele deve usar a nossa página e não a página padrão dele. Para isso, precisamos fazer mais configurações. Na classe `SecurityConfiguration` no método `configure` que recebe o `HttpSecurity` temos o seguinte código.

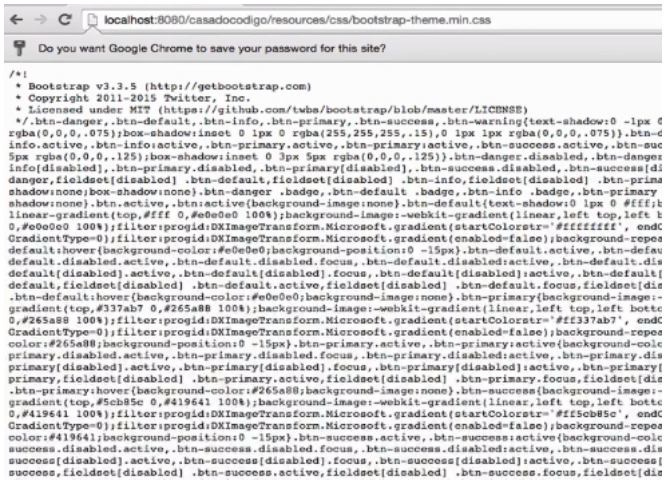
```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/produtos/form").hasRole("ADMIN")
    .antMatchers("/carrinho/**").permitAll()
    .antMatchers(HttpMethod.GET, "/produtos").permitAll()
    .antMatchers(HttpMethod.POST, "/produtos").hasRole("ADMIN")
    .antMatchers("/produtos/**").permitAll()
    .antMatchers("/").permitAll()
    .anyRequest().authenticated()
    .and().formLogin();
}
```

O objeto `HttpSecurity` tem um método chamado `loginPage` no qual passamos o caminho que será atendido pelo `LoginController` que no caso é `/login`. E neste caso precisaremos usar o `permitAll` para que o *Spring* não bloqueie a página, já que esta não está inclusa nos métodos `Matchers`. Assim teremos:

```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/produtos/form").hasRole("ADMIN")
    .antMatchers("/carrinho/**").permitAll()
    .antMatchers(HttpMethod.GET, "/produtos").permitAll()
    .antMatchers(HttpMethod.POST, "/produtos").hasRole("ADMIN")
    .antMatchers("/produtos/**").permitAll()
    .antMatchers("/").permitAll()
    .anyRequest().authenticated()
    .and().formLogin().loginPage("/login").parmitAll();
}
```

Já podemos reiniciar o servidor e tentar acessar uma das páginas bloqueadas, como por exemplo `localhost:8080/casadocodigo/produtos/` e seremos levados a nossa página de login personalizada.

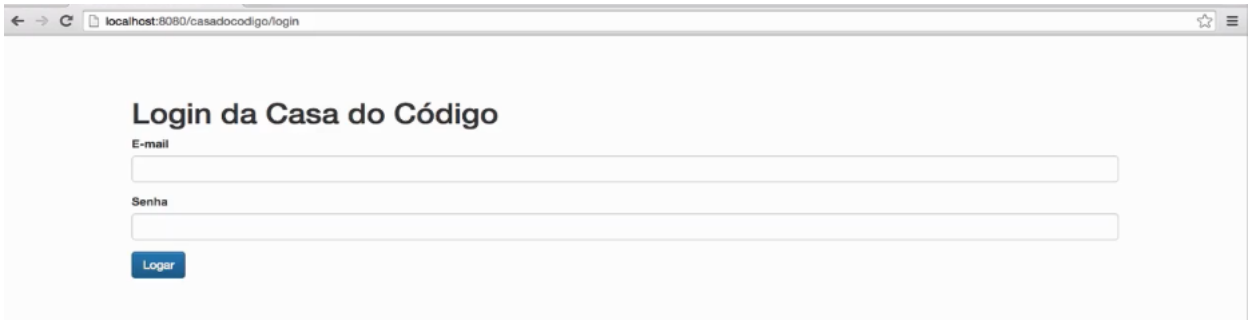
Mas ao tentarmos usar o usuário e senha que cadastramos no banco de dados, percebmos que funciona, mas há algo estranho, o código do **Bootstrap** foi impresso na página ao invés de sermos direcionados para a listagem de produtos. Mas porque?



Isso acontece porque o *Spring* redireciona para a última página ou recurso em que o acesso foi bloqueado e por incrível que pareça, deixamos as configurações do **Spring Security** bloqueando os arquivos de **CSS** e **JS** do *Bootstrap*. Precisamos usar o método `Matchers` com o valor `/resources/**` para desbloquear estes arquivos na classe `SecurityConfiguration`. Então teremos:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/produtos/form").hasRole("ADMIN")
    .antMatchers("/carrinho/**").permitAll()
    .antMatchers(HttpMethod.GET, "/produtos").permitAll()
    .antMatchers(HttpMethod.POST, "/produtos").hasRole("ADMIN")
    .antMatchers("/produtos/**").permitAll()
    .antMatchers("/resources/**").permitAll()
    .antMatchers("/").permitAll()
    .anyRequest().authenticated()
    .and().formLogin().loginPage("/login").permitAll();
}
```

Após esta alteração e reinicialização do servidor, já podemos ver visualmente alguma diferença no formulário de login.



O que falta fazermos agora para que o login funcione completamente é sua segunda parte, o **logout**. Acontece que o *logout* já está pronto, porém, por padrão, o *Spring* só aceita requisições do tipo **POST** para o *logout* que pode ser feito no **path** `/logout`.

Para deixarmos o logout um pouco mais simples, faremos com que o mesmo possa ser feito através de requisições do tipo **GET**. Assim, precisaremos apenas de mais algumas configurações na classe `SecurityConfiguration`.

No objeto `HttpSecurity` precisaremos apenas usar o método `and` para adicionamos mais uma informação na cadeia de chamadas de métodos, este que por sua vez será seguido pela chamadas dos métodos `logout` e `logoutRequestMatcher` que receberá um objeto do tipo `AntPathRequestMatcher` indicando qual o caminho no qual o *Spring* ao receber uma requisisção, deverá fazer logout do usuário corrente. O método `configure` da classe `SecurityConfiguration` deve ficar da seguinte forma:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/produtos/form").hasRole("ADMIN")
    .antMatchers("/carrinho/**").permitAll()
    .antMatchers(HttpMethod.GET, "/produtos").permitAll()
    .antMatchers(HttpMethod.POST, "/produtos").hasRole("ADMIN")
    .antMatchers("/produtos/**").permitAll()
    .antMatchers("/resources/**").permitAll()
    .antMatchers("/").permitAll()
    .anyRequest().authenticated()
    .and().formLogin().loginPage("/login").permitAll()
    .and().logout().logoutRequestMatcher(new AntPathRequestMatcher("/logout"));
}
```

Assim ao logarmos podemos ver todos os links antes ocultos e cadastrar novos produtos normalmente e para sairmos da aplicação pasta digitar `logout` na barra de endereços do navegador, sendo que o caminho completo deve ficar da seguinte forma: `localhost:8080/casadocodigo/logout`.

Recapitulando

Nesta aula, vimos como restringir partes da aplicação para usuários autenticados, como fazer o login e logout, particularidades sobre segurança e até aprendemos como evitar ataques como o *CSRF*. Fizemos também o *Spring* validar os usuários e criamos as *roles* que representam as permissões dos usuários. Aprendemos a usar algumas das tags da *security taglib* do *Spring* e vamos aprender muito mais sobre *Spring* nas próximas aulas.