04

Criando Usuario e UsuarioDAO

Transcrição

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 = :e
                .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 = :emai
            .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`.

```
@EnableWebSecurity
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 = :e
                .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, JPAConfigurat
    }

    @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',
```

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_
```

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.