05

Autenticando com os participantes do banco de dados

Até o momento, só conseguimos logar no nosso sistema com os usuários cadastrados no arquivo `shiro.ini`. O que queremos agora é que possamos fazer login, utilizando os usuários cadastrados no banco de dados.

Para isso, vamos precisar criar uma classe no pacote `br.com.caelum.auron.shiro` que implemente a interface `Realm`.

```
public class Autenticador implements Realm {

}
```

Nessa classe, precisaremos de um `ParticipanteDao` que irá procurar pelo participante no banco. Porém, o contexto do *Shiro* não se integra com CDI e portanto não podemos realizar injeções em um `Realm`. Mais a frente resolveremos este problema, por enquanto, apenas criaremos um atributo do tipo `ParticipanteDao` na classe `Autenticador`

```
public class Autenticador implements Realm {
      private ParticipanteDao participanteDao;
}
```

Precisamos agora implementar o método `getAuthenticationInfo(AuthenticationToken token)`

```
public class Autenticador implements Realm {
      private ParticipanteDao participanteDao;

      @Override
      public AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws Authent

      }
}
```

Neste método que toda mágica irá ocorrer. Caso a autenticação seja feita com sucesso, devemos retornar um objeto do tipo `AuthenticationInfo` e caso contrário, lançamos uma `AuthenticationException`.

O primeiro passo no nosso método, será verificar quais dados foram entrados pelo usuário no momento da autenticação. Isso pode ser feito através do objeto `AuthenticationToken` que recebemos como argumento. Mas antes, precisaremos fazer o *casting* para `UsernamePasswordToken`.

```
@Override
    public AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws Authentica
        UsernamePasswordToken usernameToken = (UsernamePasswordToken) token;
```

Com este objeto em mãos, temos acesso aos métodos `getUsername()` e `getPassword()` (que retorna um `char[]`). Vamos atribuir seus valores à variáveis do tipo `String`.

```
@Override
public AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws Authenticatio
        UsernamePasswordToken usernameToken = (UsernamePasswordToken) token;

        String email = usernameToken.getUsername();
        String senha = new String(usernameToken.getPassword());
}
```

Com esses dados, precisamos pegar o `Participante` do banco de dados. Para isso, vamos precisar do nosso `ParticipanteDao`. Mas como obtê-lo?

Como o contexto do Shiro não se integra com CDI, precisaremos fazer o *lookup* do `ParticipanteBean` no serviço de nomes do JBoss WildFly.

Para isso, criaremos um novo método chamado `getParticipanteDao()`:

```
public ParticipanteDao getParticipanteDAO() {
    Properties props = new Properties();
    props.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
    try {
        InitialContext ctx = new InitialContext(props);
        ParticipanteDao dao = (ParticipanteDao) ctx.lookup("java:module/ParticipanteDao");
        return dao;
    } catch(NamingException e) {
        throw new RuntimeException(e);
    }
}
```

OBS: Para mais detalhes sobre *lookup*, consulte nosso curso de **EJB**.

Vamos criar um atributo do tipo `ParticipanteDao` para fazermos um *cache* do objeto que foi obtido.

```
public class Autenticador implements Realm {
    private ParticipanteDao participanteDao;
        ...
```

Feito isso, iremos obter o `Participante` usando nosso `ParticipanteDao`

```
Participante participante = participanteDao.getParticipante(email, senha);
```

Ainda não temos o método implementar este método na classe `ParticipanteDao`, portanto vamos implementá-lo:

```
public Participante getParticipante(String email, String senha) {
    TypedQuery<Participante> query = em.createQuery("from Participante p where p.email=? and p.s
    query.setParameter(1, email);
    query.setParameter(2, senha);

    return query.getSingleResult();
}
```

Agora iremos verificar se o participante retornado não é nulo. Neste caso, vamos criar e retornar um `SimpleAuthenticationInfo`.

```
if(participante != null) {
    AuthenticationInfo info = new SimpleAuthenticationInfo(email, senha, getName());
    return info;
}
```

Caso contrário, devemos lançar uma `AuthenticationException`.

```
throw new AuthenticationException();
```

Nosso código, no final, deve ficar assim:

```
@Override
public AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws Authenticatio
    UsernamePasswordToken usernameToken = (UsernamePasswordToken) token;

    String email = usernameToken.getUsername();
    String senha = new String(usernameToken.getPassword());

    participanteDao = getParticipanteDAO();

    Participante participante = participanteDao.getParticipante(email, senha);

    if(participante != null) {
        AuthenticationInfo info = new SimpleAuthenticationInfo(email, senha, getName());
        return info;
    }
    throw new AuthenticationException();
}
```

Devemos implementar mais dois métodos: `getName()` que retorna o nome do nosso `Realm`.

```
@Override
public String getName() {
    return this.getClass().getSimpleName();
}
```

E o método `supports()` que diz se queremos suportar o *Token* recebido.

```
@Override
public boolean supports(AuthenticationToken token) {
    return true;
}
```

Agora precisamos dizer ao shiro que queremos que ele use nosso Autenticador. Para isso, abra o arquivo `shiro.ini` e faça as seguintes configurações:

```
[main]
    auronrealm = br.com.caelum.auron.shiro.Autenticador
    securityManager.realms = $auronrealm
    authc.loginUrl=/faces/login.xhtml
```

E podemos apagar o usuário definido na *Section* `[users]` .

Rode o projeto e teste o login com algum participante do banco de dados.