02

Criando um token da API e garantindo segurança

Finalmente faremos a segurança da API. No geral, a autenticação do acesso a uma API se faz com o envio de um código junto a toda requisição. Para isso, então precisamos de um código, então usaremos a mesma estratégia do **TokenDeCadastro**. Vamos criar um modelo que contenha essa informação e que se vincule a um usuário.

```
package models;
@Entity
public class TokenDaApi extends Model {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne
    private Usuario usuario;
    private String codigo;
    private Date expiracao;
    // getters
}
```

Armazenamos uma data de expiração também para caso você queira fazer um sistema de expiração do acesso após um dado tempo. Agora precisamos atribuir os valores, e faremos isso no construtor que recebe um usuário.

```
public TokenDaApi (Usuario usuario) {
    this.usuario = usuario;
    this.expiracao = new Date();
    this.codigo = Crypt.sha1(expiracao.toString()+usuario.toString()+Crypt.generateSecureCookie
}
```

Temos então um código criptografado com dados não sensíveis. Agora basta fazer a *Evolution* desta nova tabela, com o seguinte conteúdo.

```
# --- !Ups
create table token_da_api (
  id                            bigint auto_increment not null,
  usuario_id                    bigint,
  codigo                        varchar(255),
  expiracao                     datetime(6),
  constraint uq_token_da_api_usuario_id unique (usuario_id),
  constraint pk_token_da_api primary key (id)
);
alter table token_da_api add constraint fk_token_da_api_usuario_id foreign key (usuario_id) ref
# --- !Downs
alter table token_da_api drop foreign key fk_token_da_api_usuario_id;
drop table if exists token_da_api;
```

Última alteração é adicionar uma referência ao token no modelo de **Usuario**, mas sem alterar a tabela. Para isso usamos o mapeamento para a variável `"usuario"` .

```
@OneToOne(mappedBy = "usuario")
private TokenDaApi token;
// getter e setter
```

Agora, ao confirmar o cadastro, podemos gerar e salvar um token para o novo usuário.

```
public Result confirmaUsuario(String email, String codigo) {
    //...
        if (token.getUsuario().equals(usuario)) {
            //...
            TokenDaApi tokenDaApi = new TokenDaApi(usuario);
            tokenDaApi.save();
            usuario.setToken(tokenDaApi);
            usuario.update();
            //...
        }
    //...
}
```

Faça logout, remova seus usuários existentes do banco. Assim, podemos fazer uma última melhoria que garante a segurança do nosso sistema: trocar o email do usuário na sessão pelo código do token, que resolve as questões de segurança que vimos antes! Vamos aproveitar pra otimizar o código. Já que inserimos o usuário na sessão em dois momentos diferentes no **UsuarioController**, vamos extrair a lógica para reaproveitar e já inserir o código!

```
private void insereUsuarioNaSessao(Usuario usuario) {
    session(AUTH, usuario.getToken().getCodigo());
}
```

Agora como o conteúdo da sessão mudou, precisamos alterar a lógica de autenticação. Para isso, precisamos criar um método no **UsuarioDAO** que retorna um usuário pelo código do token dele e em seguida alterar a lógica do **UsuarioAutenticado** para utilizar esse método.

```
public class UsuarioDAO {
    //...
    public Optional<Usuario> comToken(String codigo) {
        Usuario usuario = usuarios
            .where()
            .eq("token.codigo", codigo)
            .findUnique();
        return Optional.ofNullable(usuario);
    }
    //...
}
```

```
public class UsuarioAutenticado extends Authenticator {
    //...
    @Override
    public String getUsername(Context context) {
        String codigo = context.session().get(UsuarioController.AUTH);
        Optional<Usuario> possivelUsuario = usuarioDAO.comToken(codigo);
```

```
        //...
    }
    //...
}
```

Enfim temos um sistema de autenticação bem seguro! Agora, já que vamos cobrar o código do usuário na autenticação da API, precisamos mostrar para ele qual o seu código de acesso no painel `painel.scala.html` . Utilizando uma estilização do *Bootstrap*, altere o conteúdo do arquivo pelo seguinte.

```
@(usuario: Usuario)
@main("Painel do usuário") {
  <section class="jumbotron">
    <h1>Painel do usuário</h1>
    <p>Bem vindo, <strong>@usuario.getNome()</strong>! Este é seu painel. Aqui você poderá ver s
  </section>
  <section class="panel panel-default">
    <header class="panel-heading">
      <h2 class="panel-title">Sua chave de acesso pessoal</h2>
    </header>
    <p class="panel-body">
      Sua chave de acesso é:
      <button class="btn" data-toggle="collapse" data-target="#tokenApi"><span class="glyphicon
      <strong class="token collapse" id="tokenApi">@usuario.getToken().getCodigo()</strong>
    </p>
    <p class="panel-body">
      Para acessar a API, faça um GET na seguinte URL (adicionando no cabeçalho "API-Token" a su
    </p>
  </section>
}
```

Utilizamos um pouco de javascript para mostrar a chave do usuário só quando ele clicar em um botão, portanto para que isso funcione precisamos lembrar de importar o javascript do *Bootstrap* na tela, e o do *jQuery* já que é uma dependência. Vamos manter os imports centralizados e fazer isso na view `main.scala.html` .

```
//...
<body>
    //...
    </main>
    <script type="text/javascript" src="@routes.Assets.versioned("bootstrap/js/jquery.min.js")"
    <script type="text/javascript" src="@routes.Assets.versioned("bootstrap/js/bootstrap.min.js'
</body>
</html>
```

E por fim altere o painel() para que ele receba o código de autenticação.

```
@Authenticated(UsuarioAutenticado.class)
    public Result painel() {
        String codigo = session(AUTH);
        Usuario usuario = usuarioDAO.comToken(codigo).get();
        return ok(painel.render(usuario));
    }
```