02

Criando um token e configurando o enviador de emails

Não fizemos nenhum tipo de confirmação da posse do email, então quando um usuário se cadastra ele pode estar criando um email falso ou tomando o de outra pessoa. Para garantir que isso não aconteça, podemos criar uma confirmação por email, que em geral consiste de um link com um código de cadastro. Vamos alterar o modelo de usuário para que este tenha um marcador indicando se ele já foi ou não confirmado e também criar um modelo de token temporário para usar na confirmação do cadastro.

```
public class Usuario ... {
    // ...
    private boolean verificado;
    public void setVerificado(boolean verificado) {
        this.verificado = verificado;
    }
    public boolean isVerificado() {
        return verificado;
    }
}
```

```
package models;
@Entity
public class TokenDeCadastro {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne
    private Usuario usuario;
    private String codigo;
    // criar os getters de cada atributo
}
```

Precisamos também gerar os modelos no banco de dados com uma nova evolução. Crie o novo arquivo com o seguinte conteúdo.

```
# --- !Ups
create table token_de_cadastro (
  id                            bigint auto_increment not null,
  usuario_id                    bigint,
  token                         varchar(255),
  constraint uq_token_de_cadastro_usuario_id unique (usuario_id),
  constraint pk_token_de_cadastro primary key (id)
);
alter table token_de_cadastro add constraint fk_token_de_cadastro_usuario_id foreign key (usuar
# --- !Downs
alter table token_de_cadastro drop foreign key fk_token_de_cadastro_usuario_id;
drop table if exists token_de_cadastro;
```

Podemos executar a evolução acessando o servidor e então criar um construtor para setar os atributos do modelo de token, utilizando criptografia com alguns dados do usuário e um código aleatório para gerar o codigo de segurança.

```
public TokenDeCadastro(Usuario usuario) {
    this.usuario = usuario;
    this.codigo = Crypt.sha1(usuario.getNome()+usuario.getEmail()+Crypt.generateSecureCookie());
}
```

E agora que já temos nosso token funcional, precisamos criar e salvar esse token após salvar o usuário, no controller de usuários.

```
public Result salvaNovoUsuario() {
    //...
    usuario.save();
    TokenDeCadastro token = new TokenDeCadastro(usuario);
    token.save();
    //...
}
```

Agora vem a parte mais complicada, configurar o email para envio. Para isso precisamos baixar mais uma dependência no `build.sbt`. Adicionamos a dependência do enviador de emails do play, o *PlayMailer*.

```
libraryDependencies ++= Seq(
  //...
  "com.typesafe.play" % "play-mailer_2.11" % "5.0.0-M1"
)
```

Com a dependência pronta para baixar, precisamos dizer ao Play! quais as configurações que serão utilizadas para enviar um email. Como estamos em desenvolvimento, usaremos dados de teste.

```
play.mailer {
  host=mock.mailer.com
  port=25
  ssl=no
  tls=no
  user=mock.user
  password=mock.password
  debug=no
  timeout=60000
  connectiontimeout=60000
  mock=yes
}
```

A configuração `mock=yes` indica que o email não deve ser enviado, somente impresso no log do servidor. Para enviar email de verdade, é necessário remover essa configuração (ou inserir o valor `no`) e utilizar configurações válidas de um servidor de emails real. Como o *Eclipse* não lida tão bem com as atualizações de dependências, atualize o projeto executando no console os comandos `reload` e `eclipse`. Se necessário, atualize o projeto no próprio *Eclipse* (atalho: `F5`).

Agora podemos enviar um email! Para isso, precisamos de um cliente de envio de emails, o MailerClient do Play!.
Podemos recebê-lo injetado e enviar um email, por enquanto vazio ao salvar o token de cadastro.

```
@Inject
private MailerClient enviador;
public Result salvaNovoUsuario() {
    //...
    token.save();
    enviador.send(new Email());
    //...
}
```

O próximo passo é criar o conteúdo do email!