

PLAY FRAMEWORK NA PRÁTICA

GASTE TEMPO NO QUE É PRECIOSO



ALBERTO SOUZA

Play Framework na prática

Gaste tempo no que é precioso

Alberto Souza

Esse livro está à venda em <http://leanpub.com/playframeworknapratica>

Essa versão foi publicada em 2014-07-31



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 Alberto Souza

Tweet Sobre Esse Livro!

Por favor ajude Alberto Souza a divulgar esse livro no [Twitter!](#)

A hashtag sugerida para esse livro é [#playframeworknapratica](#).

Descubra o que as outras pessoas estão falando sobre esse livro clicando nesse link para buscar a hashtag no Twitter:

<https://twitter.com/search?q=#playframeworknapratica>

Escrever um livro não é fácil. É necessário dedicar bastante tempo para uma tarefa que não necessariamente vai te retornar alguma coisa. Em contra partida, como professor, eu amo espalhar conhecimento e um livro é uma forma de registrar esse conhecimento e mantê-lo por prazo indeterminado!.

Pelo apoio e paciência eu queria agradecer a Larissa, minha esposa. Afinal de contas por conta desse livro ela passou menos horas comigo. Além disso, não posso deixar de agradecer ao amigo Mauricio Aniche que, inúmeras vezes, me chamou de sem foco e disse que eu não escreveria. Uma competição é tudo na vida né?

Conteúdo

Introdução	1
Seguir ou não a especificação	1
Sem pausas para recarregamento de classes	2
Pensado para ser escalável	2
Comece a aventura	3
Primeira Aplicação	4
Criando o primeiro projeto	4
Rodando pela primeira vez	5
Conclusão	6
Definindo a aplicação	7
Criando o projeto e importando para o eclipse	7
Um pouco da estrutura de pastas	9
Conclusão	9
Início do cadastro de eventos	10
Entendendo um pouco mais sobre Controllers	11
Acessando a página	13
Conclusão	15
Tratando as requisições	16
Recebendo os parâmetros do formulário	18
Conclusão	21
Gravando e listando os eventos	22
Gravando com o Ebean	22
Configurando o Ebean	22
Criando as tabelas com as Evolutions	26
Listando os eventos e o Redirect	28
Conclusão	30
Convertendo e Validando os dados	32
Pequeno problema com a Evolution do Ebean	33
Problema na conversão	34

CONTEÚDO

Criando o seu formatter	35
Validando os dados básicos	37
Html helpers para facilitar construção de formulários	40
Trocando as mensagens de validação	42
Criando uma validação reaproveitável	43
Validação específica do modelo	46
Conclusão	47
Fazendo o upload da imagem do evento	48
Lidando com upload no Controller	48
Integrando o upload com a gravação do evento	50
Conclusão	51
Enviando emails de maneira assíncrona	52
Aprovando o evento	54
Compondo urls com parâmetros	56
Enviando email	58
Problemas na execução de código síncrono	61
Suporte nativo a execução assíncrona	61
Conclusão	64
Cacheando o resultado das actions	65
Começando com cache no Play	65
Invalidando o cache programaticamente	67
Cuidado com tipo de cache que você deseja	68
Conclusão	68
Protegendo o sistema	69
Realizando o login	69
Restringindo o acesso	73
Compondo actions para evitar duplicação de código	75
Plugin de autenticação do Play	77
Conclusão	78
Servindo formatos distintos	79
Qual formato servir?	79
Problema gerado pelo @Cached	80
Conclusão	81
Testes automatizados	82
Teste de integração nos Dao's	82
O problema de limpar o banco entre os testes	86
Testes de integração para os controllers	90
Testes de aceitação	93

CONTEÚDO

Conclusão	95
Layout e reaproveitamento de views	96
Organizando os recursos estáticos	96
Criando o template do sistema	97
Melhorando a legibilidade com includes	101
Um pouco mais de Scala para as views	103
Conclusão	105
Internacionalizando a aplicação	106
Externalizando strings	106
Mudando o idioma programaticamente	109
Atenção com o Cache	111
Conclusão	111
Fazendo o deploy da aplicação	112
Executando em modo de produção	112
Configurando o servidor de produção	113
Escalando ainda mais	115
Stateless ajudando na escalabilidade	118
Https	118
Deploy no cloud	119
Conclusão	120
Até uma próxima	120
Reaproveitando funcionalidades através de plugins	121
Utilizando e implementando o plugin	121
Detalhes na implementação de um plugin	123
Publicando	124
Conclusão	125
Cuidando da evolução do banco	126
Evolutions no Play	127
Problemas comuns	128
Outras ferramentas	131
Conclusão	131
Hora de praticar	132
Mantenha contato	132
Consultoria e eventos	132

Introdução

Existem diversos frameworks MVC para desenvolver aplicações WEB e talvez você esteja se perguntando porque deveria aprender um novo? A motivação principal é tentar usar uma ferramenta que vai fazer você se preocupar apenas com suas regras de negócio.

O Play já vem com várias decisões de infra estrutura tomadas por você. Por exemplo:

- Servidor que vai ser utilizado para publicar a aplicação
- Framework de mapeamento objeto relacional
- Suporte nativo a configuração de ambientes como teste, homolog e produção
- Classes utilitárias para tornar os testes de integração e aceitação mais fáceis de serem escritos.
- Views compiladas
- Tarefas para prontas para empacotar e distribuir a aplicação
- Plugin para controlar a evolução do banco de dados e manter um histórico das criações das suas tabelas.
- Altamente performático e escalável, por conta de sua natureza assíncrona.

Perceba que toda essa infra estrutura sozinha não serve para nada e mesmo assim várias horas são gastas pelo time de desenvolvimento do projeto montando essa infra de projeto para projeto. Com toda essa parte já pronta, os programadores podem ficar preocupados a maior parte do tempo apenas com a escrita das regras de negócio relativas a sua aplicação o que, provavelmente, vai ser muito mais rentável para a empresa.

Seguir ou não a especificação

Para conseguir prover muito mais do que os concorrentes, o Play teve que tomar uma decisão arquitetural ousada. As especificações que regem o desenvolvimento de aplicações web em Java não foram pensadas para fornecer as características oferecidas pelo framework. Para não ter que ficar usando soluções de contorno, o Play decidiu não seguir as especificações do JAVAE. De início pode parecer que você perde muito, já que agora está fortemente acoplado a ele, mas pare e pense por um segundo. Qual é a frequência que você fica trocando de implementação? Quantas vezes um projeto que você estava participando teve que trocar a implementação de JPA? Ou a implementação da especificação JAX-WS? A que cuida da criação e consumo de WebServices.

Foi baseado nisso que o time por trás do Play decidiu não dar muita importância para a especificação. Indo pelo outro caminho, eles conseguiram criar um framework que te oferece várias tecnologias para diversas camadas da sua aplicação e com tudo já integrado. Estes tipos de frameworks são conhecidos no mercado como Full Stack e tem se mostrado mais produtivos do que os concorrentes. Exemplos já consolidados são o Rails que é escrito em Ruby e o Django que foi escrito em Python.

Sem pausas para recarregamento de classes

Uma outra característica ruim do desenvolvimento de aplicações web em Java é o famoso reload obrigatório que a aplicação tem que sofrer quando alguma classe é alterada. Pense em quantas vezes você ficou esperando o **hot deploy** do Tomcat após alguma alteração. E quando você estava com um usuário logado e o mesmo era perdido após a alteração. Esse é um problema recorrente que faz os desenvolvedores perderem minutos preciosos de maneira ociosa. Tanto que muitos desenvolvem em modo de **debug** o tempo todo para tentar minimizar essa espera. Mas e quando um arquivo de configuração é alterado, por exemplo algum **.properties** ou o **persitence.xml** de sua aplicação que usa JPA? Para esses não tem jeito, tem que esperar o redeploy para esperar refletir as alterações.

Como tudo no Play é integrado, esse foi problema também foi resolvido. Qualquer arquivo que você altere na sua aplicação é automaticamente recarregado pelo Play. Muito parecido com o que acontece durante o desenvolvimento usando linguagens de script, tipo Ruby e Php. Os programadores que vem da plataforma .NET também estão acostumados com isso devido ao auto grau de integração entre os frameworks da plataforma e a IDE de desenvolvimento.

Pensado para ser escalável

Um último ponto que eu quero abordar para te convencer a devorar esse livro como se seu projeto dependesse disso é o poder de escalabilidade do Play. Ainda hoje a solução mais comum para escalar uma aplicação é adicionar mais servidores e fazer um balanceamento de carga entre os servidores. Essa é uma solução que eventualmente vai ter que ser tomada, mas o que você mais quer é conseguir postergar um aumento na complexidade do projeto por conta dessa decisão. Caso a sua aplicação esteja instalada em diversos servidores diferentes, como que ela vai lidar com os dados da sessão do usuário? Esse é apenas um exemplo de complexidade adicionada.

Para tentar melhorar esse aspecto, o Play usa um servidor chamado Netty que foi construído já pensando em lidar com os requests de maneira completamente assíncrona. Mas ele foi além, pois nada adianta seu servidor ter esse suporte se suas API's não possuem o mesmo princípio. O core do Play foi construído baseado na linguagem Scala, que roda em cima da JVM e tem uma preocupação forte com concorrência e paralelismo. Além disso foi utilizado o projeto Akka, que hoje em dia é principal projeto do mundo Java quando se fala dessas características. Dessa forma, tudo dentro do framework acontece de maneira não bloqueante o que aumenta em muito a escalabilidade de apenas uma instância da sua aplicação.

E caso você ainda assim precise escalar a aplicação de maneira horizontal, colocando mais servidores, o Play transforma isso numa tarefa trivial. Ele foi projetado para não manter estado de nada, ser Stateless. Sem estado não existe motivo para se preocupar em qual servidor o request vai cair. O seu balanceamento de carga vai ser usada no máximo da capacidade.

Comece a aventura

Caso os argumentos acima tenham aguçado sua curiosidade, comece a ler o livro agora! No próximo capítulo você já vai ter um “Ola Mundo” construído em questão de minutos. E aproveite, se você acha que já tem tudo isso, volte na LeanPub e peça o reembolso, você tem um prazo de até 45 dias para isso.

Primeira Aplicação

Para começar, vamos logo criar nossa primeira aplicação com o Play!. Fique atento, pois daqui a pouco você já vai ter um **Hello World** para mostrar aos seus colegas. A primeira coisa que precisamos fazer, assim como em qualquer framework, é baixar o Play. Basta acessar o site do projeto, cujo endereço é <http://www.playframework.org/> e clicar no link de download. O download do Play, assim como a *JDK* do Java, vai muito além das classes que vão nos ajudar na escrita do projeto. Vai ser baixado um conjunto de ferramentas que vai nos ajudar a criar o projeto, iniciar e parar o servidor, empacotar o projeto entre outras funcionalidades.

Criando o primeiro projeto

Uma vez baixado, escolha o local mais apropriado para fazer a extração do *zip*. Para facilitar nossa comunicação, a partir de agora chamarei o caminho escolhido para fazer a extração de **\$caminhosdkplay**. Agora vamos criar nossa primeira aplicação, não saindo muito do caso comum, vamos chamá-la de **oimundo**. Para isso basta abrir um terminal, ir até a pasta de seus projetos e digitar o seguinte:

```
1 $caminhosdkplay/activator new oimundo
```

Lembre de substituir o **\$caminhosdkplay** pelo seu diretório de instalação. Quando a linha acima for executada, vai ser mostrado para você uma mensagem do Play fornecendo alguns modelos que podem servir de base para a criação do seu projeto.

```
Fetching the latest list of templates...

Browse the list of templates: http://typesafe.com/activator/templates
Choose from these featured templates or enter a template name:
  1) minimal-java
  2) minimal-scala
  3) play-java
  4) play-scala
(hit tab to see a list of all templates)
> █
```

Digite a opção 3, **play-java**. A escolha desse modelo vai fazer com que seja criada uma aplicação apenas com o mínimo necessário.

Uma pasta com o nome do projeto vai ser criada. De maneira simples e prática, temos um novo projeto. Entrando na pasta do projeto, pode ser percebido que o Play criou uma estrutura específica para nós. Tente segurar a curiosidade, sei que é difícil, mas não é interessante sairmos dissecando um monte de pastas sem termos uma real necessidade. O importante agora é subir o servidor e ver nossa primeira aplicação rodando.

Rodando pela primeira vez

A partir da versão 2.3, tudo que é necessário para rodar o seu projeto, já está dentro da pasta criada pelo Play. Para entrar no console do Play, de dentro da pasta do projeto, basta digitar o seguinte comando:

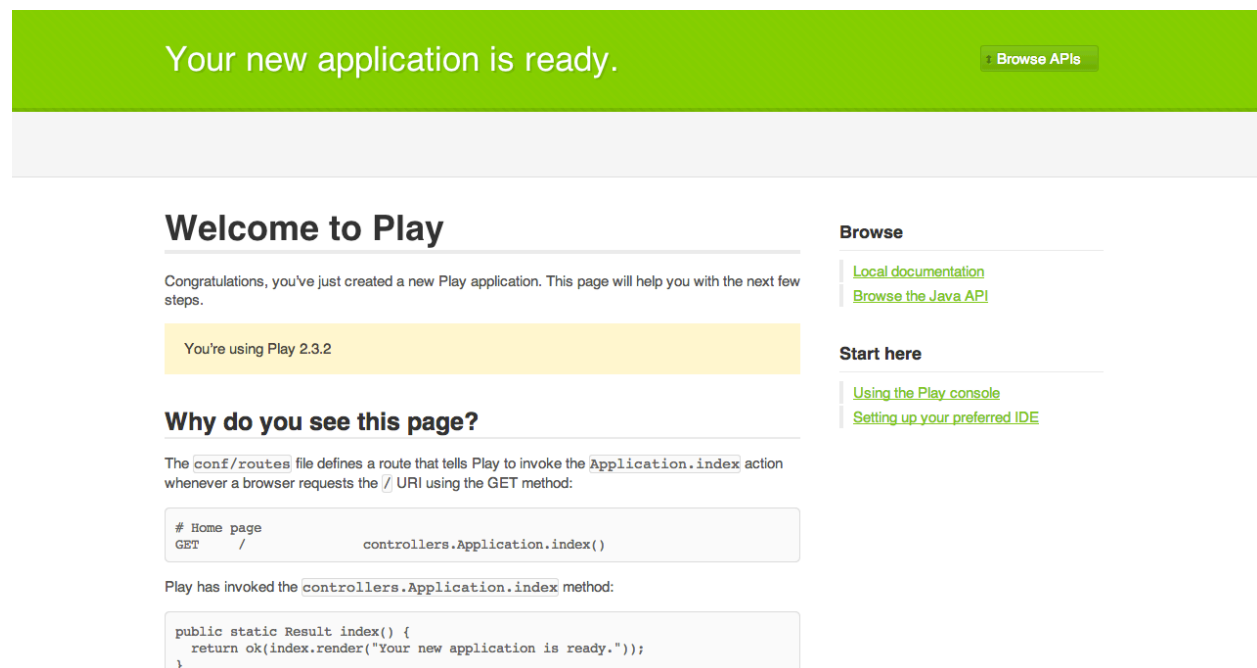
```
1 ./activator
```

Caso esteja fazendo isso pela primeira vez, você vai ser obrigado a esperar um pouco. O Play vai tentar baixar todas as dependências necessárias para que o projeto seja executado, espere todo o processo até o fim.

Após esse passo inicial, vai ser aberto no terminal o console do Play, que vai ser exibido de maneira parecida com a que segue abaixo:

```
[info] Loading project definition from /Users/albertoluizsouza/ambiente/desenvolvimento/scala/oimundo-play23/project
[info] Set current project to oimundo-play23 (in build file:/Users/albertoluizsouza/ambiente/desenvolvimento/scala/oimundo-play23/)
[oimundo-play23] $
```

Agora basta digitar o comando `run` e o servidor do Play vai ser iniciado. Repare que o servidor roda, por default, na porta 9000. Acesse a url `http://localhost:9000` e pronto! nossa primeira aplicação já está funcionando. Uma parte da página que vai ser exibida é a que segue abaixo:



Welcome to Play

Congratulations, you've just created a new Play application. This page will help you with the next few steps.

You're using Play 2.3.2

Why do you see this page?

The `conf/routes` file defines a route that tells Play to invoke the `Application.index` action whenever a browser requests the `/` URI using the GET method:

```
# Home page
GET    /      controllers.Application.index()
```

Play has invoked the `controllers.Application.index` method:

```
public static Result index() {
    return ok(index.render("Your new application is ready.));
}
```

Browse

- [Local documentation](#)
- [Browse the Java API](#)

Start here

- [Using the Play console](#)
- [Setting up your preferred IDE](#)

Para parar o servidor basta pressionar **Ctrl+D** e, para sair do console, digite `exit`.

Colocando o caminho do sdk no Path

Toda vez que necessitamos criar um novo projeto, é necessário digitar o tempo todo `$caminhosdk-play/activator`. Para isso precisamos acessar o caminho de instalação do sdk. Ficar digitando sempre esse caminho tende a ficar muito chato. Para facilitar, comumente adicionamos esse caminho ao *path* do nosso sistema para que fique mais simples. Em sistemas Unix esse processo é tão simples quanto executar o comando abaixo:

```
1 export PATH=$PATH:$caminhosdkplay
```

Não podemos esquecer de que o arquivo `$caminhosdkplay/activator` deve ser executável:

```
1 chmod a+x $caminhosdkplay/activator
```

Nas versões anteriores, o executável chama-se **play** ao invés de **activator**.

Conclusão

Nesse capítulo construímos nosso primeiro esboço de aplicação, espero que sem muitas dificuldades. A partir dos próximos capítulos vamos começar a construir uma aplicação real e, de verdade, não vá dormir ainda, tem muitas emoções vindo nas próximas cenas.

Definindo a aplicação

Agora que já fizemos um primeiro exemplo e estamos mais familiarizados com o fluxo de criação de projetos, vamos partir para a construção da aplicação que vamos usar durante o livro. O nosso projeto vai ser baseado no Agendatech, uma aplicação escrita usando o framework Rails, que tenta mapear eventos que estão para ocorrer no Brasil. O interessante de usarmos o Agendatech como base, é que já temos um domínio conhecido onde é possível ver todo o potencial que o Play tem para nos oferecer.

Para dar um gostinho, vamos adiantar as funcionalidades previstas para serem implementadas durante a leitura do livro. Lembre-se, para cada problema que for aparecendo vamos usar uma facilidade do Play para nos ajudar a resolver. Abaixo segue a lista:

- Cadastro de eventos que estão para ocorrer
- Cadastro das categorias dos eventos
- Listagem dos eventos
- Upload do logo do evento em diferente formatos
- Aprovação do evento
- Email de aprovação
- Melhora na performance e escalabilidade da aplicação
- Servir os eventos para uma aplicação terceira, por exemplo mobile.
- Login pelo facebook e twitter

A tela inicial do nosso projeto deve ficar parecida com essa:



Criando o projeto e importando para o eclipse

O primeiro passo é a criação do projeto, para isso vamos usar o sdk que baixamos. Como já colocamos o sdk no path do nosso sistema operacional, basta escolher a sua pasta preferida e rodar a seguinte linha de comando pelo *terminal* `activator new agendatech`

Agora estamos com a estrutura do projeto criada. Um detalhe que não foi discutido até aqui é a *IDE* que vamos utilizar para desenvolver. Por facilidade, vamos escolher o Eclipse. Esse passo algumas vezes pode dar uma travada no nosso fluxo. Quem nunca perdeu um tempo tendo que ajustar o *classpath*, ou porque um dos arquivos de configuração da IDE não veio junto no projeto. Para facilitar essa parte, o Play já traz junto dele uma task de importação do projeto para o eclipse. O primeiro passo é acessar a pasta onde o projeto foi criado. Agora, na linha de comando, basta executar a seguinte instrução:

```
1 ./activator
```

Isso vai abrir o console de comandos do play. Caso queira dar uma olhada em tudo que está disponível, basta digitar o seguinte:

```
1 help
```

Uma listagem dos comandos disponíveis vai ser exibida. Abaixo segue o exemplo:

```
help          Displays this help message or prints detailed help on requested commands (run 'help <command>').
completions  Displays a list of completions for the given argument string (run 'completions <string>').
about        Displays basic information about sbt and the build.
tasks        Lists the tasks defined for the current project.
settings     Lists the settings defined for the current project.
reload       (Re)loads the project in the current directory.

reload plugins (Re)loads the plugins project (under project directory).

reload return (Re)loads the root project (and leaves the plugins project).

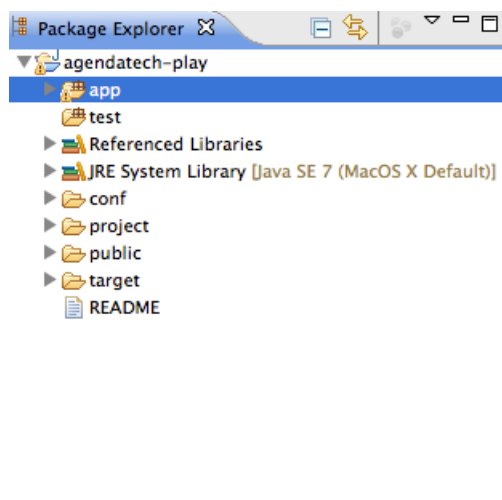
projects      Lists the names of available projects or temporarily adds/removes extra builds to the session.
project       Displays the current project or changes to the provided 'project'.
set [every] <setting> Evaluates a Setting and applies it to the current project.
session       Manipulates session settings. For details, run 'help session'.
inspect [uses|tree|definitions] <key> Prints the value for 'key', the defining scope, delegates, related definitions, and dependencies.
<log-level>   Sets the logging level to 'log-level'. Valid levels: debug, info, warn, error
plugins       Lists currently available plugins.
; <command> (; <command>)* Runs the provided semicolon-separated commands.
~ <command>    Executes the specified command whenever source files change.
last          Displays output from a previous command or the output from a specific task.
last-grep     Shows lines from the last output for 'key' that match 'pattern'.
export <tasks>+ Executes tasks and displays the equivalent command lines.
exit          Terminates the build.
---<command>  Schedules a command to run before other commands on startup.
show <key>    Displays the result of evaluating the setting or task associated with 'key'.
all <task>+   Executes all of the specified tasks concurrently.

More command help available using 'help <command>' for:
!, +, ++, <, alias, append, apply, eval, iflast, onFailure, reboot, shell
```

Um dos comandos é justamente o *eclipse*, que nos permite criar toda estrutura necessária para importar o projeto para o Eclipse. Sem perder muito tempo, aproveite e já importe a aplicação. Para isso digite o comando no console, como demonstrado abaixo:

```
1 eclipse
```

Agora basta importar o projeto para seu eclipse. Após a importação, a estrutura do seu projeto deve ficar parecida com a que segue abaixo:



Para ver se está tudo certo, execute o comando *run* no terminal para colocar aplicação no ar. Se deu tudo certo, deve aparecer a tela de boas vindas padrão do Play. Agora vamos dar uma rápida parada no servidor, para isso pressione *ctrl + d*.

Um pouco da estrutura de pastas

Para o leitor mais curioso, deve ter chamado a atenção que dentro das pastas *project* e *target* existem arquivos da linguagem *Scala*. Não se preocupe, o *core* do Play2 é escrito em *Scala*, mas durante quase todo desenvolvimento vamos lidar apenas com *Java*.

Um outro ponto interessante é a estrutura de pacotes que o Play gerou para a gente. Como foi debatido antes, ao invés de ficar gastando tempo em como separar esse tipo de coisa, o framework já gerou para a gente uma *source folder* chamada **app**, com os seguintes pacotes:

- controllers
- views

Não se preocupe muito com eles agora, no próximo capítulo já vamos começar a desenvolver as primeiras funcionalidades da aplicação e, com certeza, eles entrarão em cena!

Conclusão

Neste capítulo foi apresentado a ideia que temos para desenvolver o projeto durante o livro. Vamos utilizar o Agendatech pois ele possui diversas funcionalidades que são comuns em vários projetos. E é com isso que vamos nos preocupar, com as regras de negócio! Detalhes de infraestrutura ficam para o framework.

Início do cadastro de eventos

Agora que o projeto já foi criado, é hora de dar início a implementação das funcionalidades em si. A primeira coisa a ser feita é a tela de cadastro de novos eventos. Para isso já vamos começar a usar a estrutura fornecida pelo Play. A tela não tem nada demais, um simples html.

```
1 <html>
2   <body>
3     <form action="/eventos" method="POST">
4       <input type="text" id="nome" name="nome" value="" >
5       <input type="text" id="emailParaContato" name="emailParaContato" value="" >
6       <textarea id="descricao" name="descricao" ></textarea>
7       <input type="text" id="site" name="site" value="" >
8       <input type="submit" value="Novo evento">
9     </form>
10  </body>
11 </html>
```

Por enquanto nossa **view** não tem nada demais, html simples. Um arquivo com o nome **novo.scala.html** vai ser criado e colocado dentro do pacote **app/views/eventos**. E aqui já entra alguns detalhes do Play. O primeiro é a palavra **scala** no nome do arquivo. O Play utiliza esta linguagem para possibilitar a escrita de páginas dinâmicas. Encare como mais uma tecnologia de view que você já estudou. Vamos só lembrar de algumas famosas:

- JSP. Muito utilizada em conjunto com os frameworks Java do mercado
- Velocity. Já foi uma alternativa poderosa aos JSP's.
- Freemarker. Também bastante utilizada devido a seu mecanismos de templates.
- ERB. Utilizada no mundo Ruby, principalmente com o framework Rails.

O uso da linguagem Scala está para a view do Play assim como a *Expression Language* está para a JSP. Ela possui alguns pontos muito interessantes, os quais serão descobertos na medida que as funcionalidades forem sendo desenvolvidas. Um outro ponto a se notar é que foi dito para criar a página dentro do pacote **views/eventos**. O pacote **views** é o que o Play usa como convenção para guardar as páginas. Todas as páginas criadas dentro dele recebem um tratamento especial do framework. Um que já vai ser utilizado é o fato dele transformar a página numa classe que pode ser acessada de qualquer lugar. Dentro do views você é liberado para criar a estrutura que quiser. Durante o livro as convenções do Play serão respeitadas e o motivo é que não queremos perder tempo com infraestrutura!.

O ponto alto de ter uma view que é compilada é que não é necessário esperar rodar a página para descobrir um erro. Para ir acompanhando se tudo que está sendo desenvolvido está compilando, acesse o terminal do seu computador, na pasta do seu projeto e digite o comando **play**. Depois que o console do play for aberto, digite **~ run**. O comando run, como já foi visto, faz o servidor ser iniciado e o **~** na frente é para o Play ficar recompilando quaisquer alterações e aplicando-as no projeto. Dessa forma qualquer erro já é exibido no terminal. Veremos que no Play quase tudo é compilado, não apenas as suas classes Java.

Entendendo um pouco mais sobre Controllers

O pensamento inicial pode ser que para acessar essa página deve-se escrever o nome do arquivo no navegador e pronto. Lembre, sua página pode conter lógicas como:

- listagem de eventos
- exibição de mensagens de sucesso e erro
- recuperação de valores para serem exibidos nos campos dos formulários

Caso o acesso fosse direto, provavelmente toda essa lógica seria colocada dentro da view, o que acarretaria em sérios problemas de manutenção. Estaríamos misturando muitas responsabilidades e escrevendo código Java num lugar inapropriado. Pensando nisso, o Play proíbe o acesso direto a qualquer view. A ideia é que você sempre passe por uma classe, que pode recuperar alguns objetos que sejam necessários para a página, e que essa classe tome a decisão sobre qual view sera chamada. Essas classes são chamadas de **Controllers**. Perceba que o próprio nome já indica que ela controla o fluxo de execução da sua aplicação web. Os controllers criados na aplicação, seguindo a convenção, vão ser colocados no pacote **controllers**. Assim como o pacote de views, seguiremos a convenção para não perder tempo configurando detalhes desnecessários.

```
1  import play.mvc.*;
2  public class EventosController extends Controller {
3
4      public static Result novo() {
5          return null;
6      }
7  }
```

Um detalhe interessante em relação aos imports que serão utilizados no decorrer do livro. Como o Play suporta o desenvolvimento em Scala ou em Java, eles tiveram que fazer uma divisão nos imports. Fique atento que qualquer import com o pacote **api** é do mundo Scala. O uso das classes Scala no seu código Java deve ser feito sempre com muita análise. O pacote usado para o mundo Java é o **play.mvc**.

Basta herdar de **Controller** e sua classe já vai ganhar diversos métodos prontos para tratar as requisições web. Aproveitamos e já adicionamos um método cujo objetivo é direcionar o usuário para a página de novos eventos.

O retorno do método *novo* é um objeto do tipo *Result*. Essa é a classe que representa o seu **Response** num projeto usando o Play. Através dela poderemos retornar diferentes status e o tipo de conteúdo que será devolvido ao cliente da nossa aplicação. Se quiséssemos retornar apenas 200, indicando que tudo deu certo, teríamos o seguinte código:

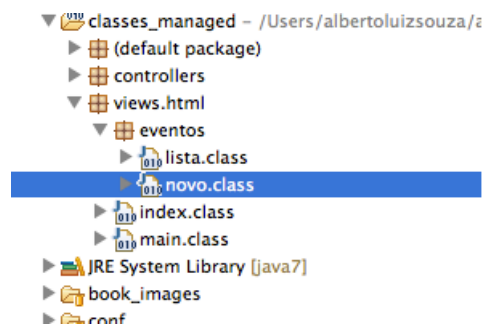
```
1  import play.mvc.*;
2  public class EventosController extends Controller {
3
4      public static Result novo() {
5          return ok();
6      }
7  }
```

O método *ok()* é disponibilizado pela super classe *Controller*. O detalhe é que não queremos retornar apenas esse status, é necessário devolver a página que será exibida no navegador. Para isso o método *ok* tem uma variação que recebe um objeto do tipo *Content*. Existem várias implementações para a interface *Content*, a que vamos usar é a que representa um retorno html, a classe *HTML*.

Nesse momento o Play facilita muito a nossa vida. Como foi dito anteriormente, toda view escrita é automaticamente compilada para uma classe scala. Conseguimos acessa-la para invocar o método *render* e obter um objeto do tipo *Html*.

```
1  import play.mvc.*;
2  public class EventosController extends Controller {
3
4      public static Result novo() {
5          Html view = views.html.eventos.novo.render();
6          return ok(view);
7      }
8  }
```

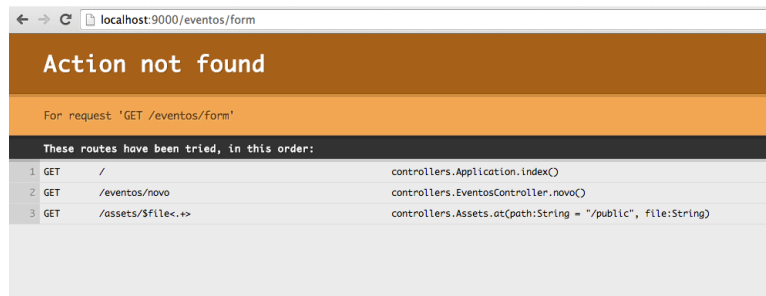
Você deve estar se perguntando de onde veio a classe, com nome minúsculo, **novo**. Para saciar a curiosidade, dê uma olhada na **source folder** chamada **classes_managed**, que foi gerada pelo Play no momento da criação do projeto.



Essa classe possui um método estático chamado *render* que retorna para gente o objeto `Html` contendo o conteúdo da nossa página. Essa é uma das grandes ideias do Play, compilar a página faz com que erros bobos como uso de variável com nome errado ou mesmo erro de sintaxe sejam pegos sem que seja necessário acessar a página. Quando nossa página ficar um pouco mais dinâmica, vamos conseguir ver ainda mais benefícios que essa feature traz para a gente.

Acessando a página

Agora que já existe o método do controller que leva para a página, falta o mais importante, o acesso a tela em si. Por exemplo, queremos que o usuário digite o endereço `http://localhost:9000/eventos/novo` para que o mesmo tenha acesso a nossa tela. Vamos ver o que acontece se fizermos isso agora:



O Play gerou uma tela de erro bem amigável para o desenvolvedor, avisando que a URL digitada ainda não foi configurada. O que falta é associar o endereço solicitado com um método de algum controller, também chamado de *Action* pelo Play. Essa configuração é realizada no arquivo **routes**, que fica na pasta **conf** do projeto. Essa pasta, como o próprio nome diz, contém todos os arquivos de configuração necessários para a nossa aplicação.

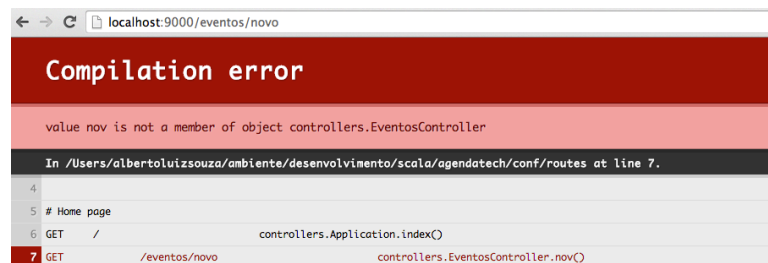
O arquivo **routes** é um arquivo de texto simples, onde mapeamos as relações entre as urls do sistemas e as actions que tratam elas.

```
1 # Routes
2 # This file defines all application routes (Higher priority routes first)
3 # ~~~~
4
5 GET      /eventos/novo      controllers.EventosController.novo()
6
7 # Map static resources from the /public folder to the /assets URL path
8 GET      /assets/*file      controllers.Assets.at(path="/public", file)
```

Perceba que ele tem uma formatação específica. A primeira parte indica qual o verbo que deve ser utilizado para acessar a action. As opções mais comuns são *POST* e *GET* mas qualquer outro suportado pode ser especificado. Depois vem a url que está sendo configurada. Ainda falaremos mais sobre essa url, porque configurações adicionais podem ser feitas. Por fim vem qual é action que vai tratar a requisição para essa url. Agora que tudo está configurado, basta acessar a url e a página será exibida normalmente.

Perceba que fizemos tudo isso sem parar o servidor nenhuma vez. Rodamos lá no início com `~ run` e o Play se encarrega de ir atualizando tudo para gente.

Agora vamos fazer um experimento. Não é incomum escrever o nome da classe errado ou errar no nome do método. Caso isso aconteça mais uma tela de erro bem amigável será exibida.



O arquivo de rotas também é compilado!. Como já foi dito, quase tudo no Play é compilado. O mesmo erro já poderia ter sido pego pelo próprio console. Já que para qualquer alteração que fazemos, o Play compila e atualiza tudo para gente com a aplicação já rodando.

```

$ sbt --batch -x11 scala sources and 1 java source to /Users/abtertuizousa/ambiente/desenvolvimento/scala/agedatech/target scala-2.10/classes...
[error] /Users/abtertuizousa/ambiente/desenvolvimento/scala/agedatech/target scala-2.10/classes... value nov is not a member of object controllers.EventsController
[error] /Users/abtertuizousa/ambiente/desenvolvimento/scala/agedatech/target scala-2.10/classes... value nov is not a member of object controllers.EventsController
[error] GET /events/novo controllers.EventsController
[error] two errors found
[error] (compile:compile) Compilation failed

```

O erro de compilação foi apenas um detalhe que a compilação nos ajudou. Ainda vamos precisar utilizar a classe gerada para fazer referência as urls. Nem perca tempo parando para lançar, já pule para o próximo capítulo para continuarmos o cadastro do evento. Um detalhe você pode ter achado

estranho é que não foi criada ainda a classe Evento. Não se preocupe, como não precisamos dela nesse momento, a mesma foi deixada para daqui a pouco.

Conclusão

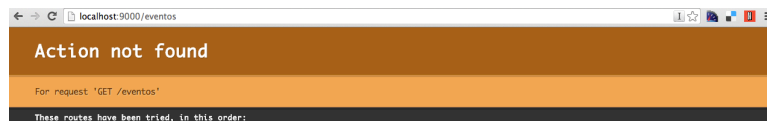
Nesse capítulo foi estudado um pouco da estrutura geral do Play. Foi construída uma view, um controller e fizemos o binding(ligação) da url com o controller através do arquivo de rotas. Vimos que o Play é fortemente baseado em convenções também. Tudo já tem seu lugar definido e dessa maneira o framework conseguiu nos ajudar bastante, fazendo com que mantenhamos o foco nas regras de negócio e não na infraestrutura. Além disso, o fato de quase tudo ser compilado virou um diferencial, pois erros são pegos mais cedo e com mensagens bem mais amigáveis do que as contidas em *StackTraces* geradas por erros de execução. Tudo isso ainda vai ser mais dissecado de acordo com as necessidades do Agendatech!.

Tratando as requisições

Agora que já existe o formulário de cadastro e os usuários já conseguem navegar até ela, o sistema precisa ser capaz de gravar um novo evento. Primeiro, é válido relembrar como está a view do formulário.

```
1  <html>
2    <body>
3      <form action="/eventos" method="POST">
4        <input type="text" id="nome" name="nome" value="" >
5        <input type="text" id="emailParaContato" name="emailParaContato" value="" \
6      >
7        <textarea id="descricao" name="descricao" ></textarea>
8        <input type="text" id="site" name="site" value="" >
9        <input type="submit" value="Novo evento">
10     </form>
11   </body>
12 </html>
```

Perceba que a action do formulário aponta para o endereço **eventos**. Provavelmente a curiosidade já apareceu e você chegou a clicar no botão de cadastro. Vamos só lembrar o que acontece.



Já tínhamos passado por esse problema quando o usuário tentou acessar a view de cadastro. É necessário fazer o binding da url com a action responsável por tratar o request. Vamos usar novamente o arquivo **routes**.

```
1  #rotas antigas
2  POST          /eventos          controllers.EventosController.cria()
```

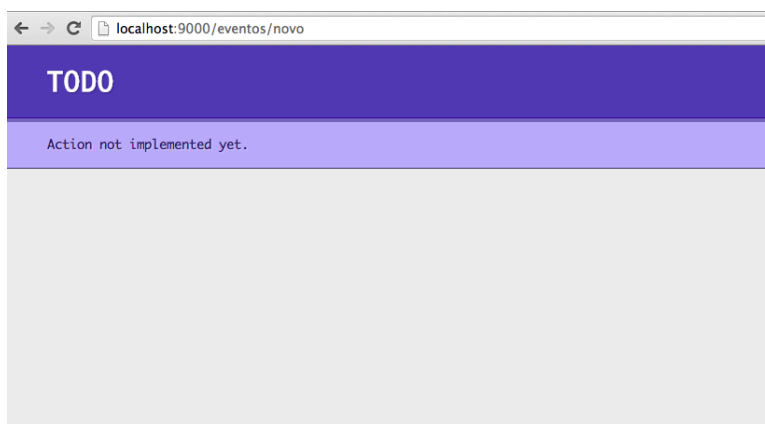
Caso tenha seguido esse passo, vai perceber que console do play acusou um erro de compilação, pois ainda não foi criado a action *cria* no controller *EventosController*.

```
1  import play.mvc.*;
2  public class EventosController extends Controller {
3
4      public static Result novo() {
5          return ok(views.html.eventos.novo.render());
6      }
7
8      public static Result cria() {
9          //o que vamos fazer aqui?
10     }
```

Como ainda não evoluímos para implementar o código da action nova, podemos usar um *Result* especial do Play que indica que está faltando a implementação do código.

```
1  public static Result cria() {
2      return TODO
3  }
```

A constante **TODO** contém um objeto do tipo *result* que gera um response com o status 501, indicando que aquela url ainda não teve sua lógica implementada. Junto com o status, é retornado também uma tela de erro seguindo o layout padrão do Play para sinalizar problemas na aplicação.



Um outro ponto que ainda pode ser melhorado antes da implementação do cadastro em si, é a maneira com que o formulário da view se refere a url da action responsável pela funcionalidade. Perceba que no atributo *action* do form, é colocado a url de maneira *hard coded*.

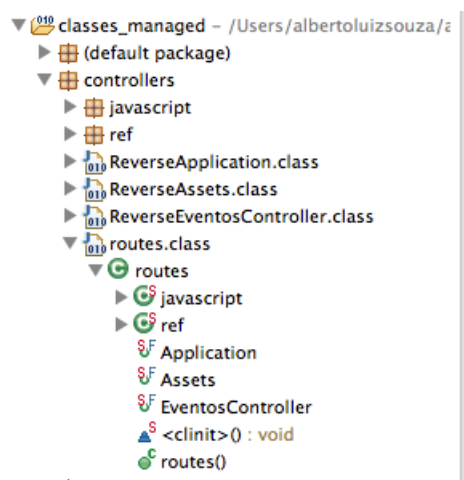
```
1  <form action="/eventos" method="POST">
```

Caso alteremos essa url no arquivo de rotas o erro só será descoberto quando alguém tentar cadastrar um evento. Não que seja um problema muito grave, pois provavelmente isso seria testado antes, mas lembre-se: quanto mais erros forem pegos em tempo de compilação, mais rápido conseguimos evoluir a aplicação. Para ajudar o programador nessa tarefa, o Play disponibiliza um objeto com métodos para acessarmos todas as rotas definidas de maneira estática.


```
1 <form action="@routes.EventosController.cria()" method="POST">
```

Para executar código dinâmico nas views do Play é necessário o uso do `@` na frente da variável. Caso tente fazer um paralelo, o leitor pode lembrar do `<? usado no PHP ou do ${} usado nas JSP's. Perceba que é disponibilizada uma variável chamada routes.`

Esse objeto é o que Play chama de rota reversa(Reverse Router). Podemos usa-lo em todas as views para que seja possível acessar todas as rotas definidas pela aplicação. Para não parecer que é uma mágica, basta conferir código gerado no source folder *classes_managed*.



Caso um erro de digitação seja cometido, o console do Play acusará um erro de compilação e o erro pode ser corrigido rapidamente. O mesmo vai acontecer se o usuário tentar acessar a aplicação por ter esquecido de olhar o console.



Recebendo os parâmetros do formulário

Precisamos começar a implementar a lógica do cadastro de eventos. A primeira coisa a ser feita é receber os parâmetros enviados através do formulário. Para isso é disponibilizado um objeto que contém todas as informações do request.

```
1 public static Result cria() {  
2     Request request = request();  
3     return TODO;  
4 }
```

O método `request` foi herdado da classe *Controller*. O mesmo possui diversos métodos para interação com o request corrente da aplicação. Agora precisamos pegar os parâmetros que foram enviados na requisição. Por exemplo, vamos pegar o nome do evento.

```
1 public static Result cria() {  
2     Request request = request();  
3     System.out.println(request.body().asFormUrlEncoded().get("nome")[0]);  
4     return TODO;  
5 }
```

Repare como é um código bem extenso. Muito longe da simplicidade de se pegar um parâmetro quando é utilizada a API de Servlets do Java.

```
1 request.getParameter("nome")
```

A motivação para tal complexidade é que o Play já foi pensado desde o início para ser um framework que pode ser utilizado tanto para uma aplicação web padrão, que realmente vai receber valores de um formulário, ou para uma aplicação que funcione como serviço. O objeto do tipo `request`, já possui métodos para receber valores em JSON e XML por exemplo.

```
1 request.body().asFormUrlEncoded();  
2 request.body().asJson();  
3 request.body().asXml();
```

Os três métodos que foram exibidos acima, são os responsáveis por tratar os diferentes formatos de dados. Ainda ficou faltando o método que temos que usar quando arquivos são enviados pelo formulário. O primeiro é o mais comum de todos, trata o formato enviado pelos formulários que escrevemos nas páginas. Perceba que o cliente pode escolher entre todos esses formatos quando vai enviar uma informação para o servidor. Isso é chamado de **Content-Type** e vem especificado pelo header **Accept** no request HTTP.

Só que ficar pegando um parâmetro de cada vez é muito trabalhoso. Na verdade se fosse obrigatório tratar cada um separadamente, provavelmente nem usaríamos esse framework. Para facilitar esse trabalho o Play já oferece um objeto, cuja classe chama *Form*, que é capaz de receber os parâmetros e nos devolver um objeto que faça mais sentido para a nossa aplicação. Por debaixo dos panos, ele usa um projeto do framework Spring, chamado **spring databinder**.

```
1  public class EventosController extends Controller {
2
3      private static Form<Evento> eventoForm = Form.form(Evento.class);
4
5      public static Result cria() {
6          Form<Evento> formFromRequest = eventoForm.bindFromRequest();
7          Evento evento = formFromRequest.get();
8          return TODO;
9      }
10 }
```

Repare que de posse do objeto *Form*, basta invocarmos o método *bindFromRequest* que ele retorna um novo *Form* com os valores do request associados ao nosso objeto. Depois basta utilizar o método *get* para recuperar o objeto de fato. O form ainda contém outros métodos, por exemplo para verificar erros de validação. Vamos lidar com eles mais para frente.

Um detalhe interessante é que finalmente precisamos da classe *Evento*. Ela não tem nada demais, por enquanto é uma classe simples com atributos privados e os métodos de acesso. Um clássico **Java Bean**.

```
1  public class Evento {
2
3      private Integer id;
4      private String emailParaContato;
5      private Estado estado;
6      private String descricao;
7      private String site;
8      private String twitter;
9      private String nome;
10
11     public void setNome(String nome){
12         this.nome = nome;
13     }
14
15     public void setDescricao(String descricao){
16         this.descricao = descricao;
17     }
18
19     //outros setter e getters
```

Perceba que os nomes dos campos do formulário são iguais aos nomes que damos aos nossos setters. É dessa maneira que o binding é feito pelo Play. O objeto *Form* ainda pode ser mais esperto. Ele consegue fazer o binding entre um *Map* simulando os parâmetros e seu objeto também. Então, caso

o input dos dados não seja via formulário e sim por um arquivo de texto que deva ser processado, podemos pegar cada linha do arquivo, gerar um mapa e depois gerar nossos objetos com facilidade.

```
1 public static Result cria() {  
2     //podemos ler o arquivo  
3     Map<String,String> parametros = //monta um hashmap baseado nas linhas  
4     Evento evento = eventoForm.bind(parametros).get();  
5     return TODO;  
6 }
```

O leitor mais atento, deve estar se questionando o porquê dos métodos dos controllers, as actions, serem estáticas. A maioria dos frameworks do mercado, tratam os controllers como classes normais, que podem ter ciclo de vida, variáveis de instância, etc. Já o Play usou uma outra abordagem. Para ele, o controller nada mais é um do que um transformador dos valores que vem da requisição HTTP para objetos que façam sentido para o domínio da aplicação. Baseado nessa premissa, eles acreditam que o controller não deveria guardar estado de nada, por isso a abordagem estática.

Esse também foi o motivo de termos declarado o atributo **eventoForm** como estático. Uma outra pergunta que pode vir a cabeça é: Se o atributo é estático, como que o Play faz para diferenciar os requests que vem de diversos clientes? A resposta é que ele usa uma classe muito comum em projetos Java que precisam disponibilizar objetos estáticos associados com uma Thread(o cliente). Essa classe é a *ThreadLocal*!

Conclusão

Nesse capítulo vimos como aproveitar a compilação do nosso arquivo de rotas para referenciar as urls nas views da nossa aplicação. Além disso, demos um passo importante, pois agora somos capazes de lidar com valores de qualquer formulário e transforma-los para os objetos do nosso domínio. Para finalizar, vimos como deixar nossa action respondendo um status que indica que tal funcionalidade ainda não foi implementada com o uso da constante **TODO**.

Gravando e listando os eventos

Agora que o objeto do tipo *Evento* já é construído baseado nas informações enviadas pelo formulário, chegou a hora de gravá-lo no banco de dados. E aqui já vamos partir para o uso direto de um framework de persistência. A opção padrão do mercado é pelo Hibernate. Só que vamos aderir a opção sugerida pelo Play, e melhor suportada por ele, que é o uso do framework Ebean. A ideia é basicamente a mesma do Hibernate, inclusive o Ebean suporta as anotações da JPA para fazer o mapeamento Objeto Relacional. A diferença maior está focada na maior facilidade de uso da API. Problemas clássicos do Hibernate como Lazy Load, fraco suporte a paginação e construções de queries parciais, aquelas que só queremos trazer parte das informações do objeto, foram resolvidas pelo Ebean.

Gravando com o Ebean

O nosso objetivo é gravar o evento no banco. Então para matar logo essa curiosidade vamos começar já mostrando o código necessário para realizar essa tarefa.

```
1  public static Result cria() {
2      Form<Evento> formFromRequest = eventoForm.bindFromRequest();
3      Evento evento = formFromRequest.get();
4      Ebean.save(evento);
5      return TODO;
6  }
```

A classe *Ebean* é ponto de entrada do framework. Ela é similar ao *EntityManager* da JPA ou a *Session* caso o leitor use Hibernate diretamente. A diferença maior é que ela não mantém nenhuma informação sobre o objeto passado, por isso que os métodos são estáticos.

Configurando o Ebean

O próximo passo é configurar as informações de acesso ao banco. No nosso caso vamos utilizar o *MySQL*, apenas por ele ser muito conhecido e de fácil instalação. O primeiro passo é adicionar as anotações necessárias na classe *Evento*.

```
1  @Entity
2  public class Evento {
3
4      @Id
5      @GeneratedValue
6      private Integer id;
7      private String emailParaContato;
8      @Enumerated(EnumType.STRING)
9      private Estado estado;
10     @Column(columnDefinition = "text")
11     private String descricao;
12     private String site;
13     private String twitter;
14     private String nome;
15
16     //getter e setters
17 }
```

Repare que não tem nada demais. Se você já conhece essas annotations pode pular para o próximo parágrafo. Abaixo segue uma lista dos significados das annotations utilizadas.

- `@Entity`. Sua classe vai representar uma tabela no banco de dados
- `@Id`. O atributo anotado é a chave primária da tabela
- `@GeneratedValue`. A geração da chave vai ser feita de maneira incremental a ser decidido pelo Ebean. No caso do MySQL vai ser utilizada o auto-increment mesmo.
- `@Enumerated`. A coluna gerada vai ser do tipo e os valores guardados vão ter os mesmos nomes das constantes da Enum.
- `@Column`. Informações adicionais para a geração da tabela. No caso da descrição foi informado que a coluna deve ser do tipo texto

Configuração de acesso ao banco

Agora que a classe está devidamente anotada precisamos adicionar as informações de acesso ao banco. Tudo deve ser no arquivo **application.conf** que está localizada na pasta **conf**. Pasta que já é conhecida devido ao uso do arquivo **routes**. Lembre-se, nessa pasta vão as suas configurações.

```

1  application.secret="Qxk/1;Pg20UUrsuC:1]hD3Mv`KmL17TLu400Sh^3d6jio2/TLeb67EM@MaR\
2  <=A<@"
3  # The application languages
4  # ~~~~~
5  application.langs="en"
6
7  # Database configuration
8  # ~~~~~
9  # You can declare as many datasources as you want.
10 # By convention, the default datasource is named `default`
11 #
12 db.default.driver=org.h2.Driver
13 db.default.url="jdbc:h2:mem:play"
14 db.default.user=sa
15 db.default.password=""

```

Repare que configurações de língua, token para geração de cookies(vamos ainda discutir), as configurações de banco entre outras, vão todas nesse arquivo. Vamos aprendendo mais coisas com o decorrer do livro. A parte importante para a gente agora são as informações de acesso ao banco. É necessário passar o driver, url, usuário e senha de acesso.

```

1  db.default.driver=com.mysql.jdbc.Driver
2  db.default.url="jdbc:mysql://localhost/agendatech_play"
3  db.default.user=root
4  db.default.password=""

```

Configuração de scan do Ebean

Um outro passo necessário é indicar os pacotes que o Ebean deve procurar por classes anotadas. Dessa maneira ele vai conseguir carregá-las e identificar algum problema de configuração, como uma anotação usada de maneira errada por exemplo. Tudo isso ainda no **application.conf**

```

1  # Ebean configuration
2  # ~~~~~
3  # You can declare as many Ebean servers as you want.
4  # By convention, the default server is named `default`
5  #
6  ebean.default="models.*"

```

Um detalhe interessante, mas não muito comum nas aplicações de hoje em dia, é que o Ebean suporta que você configure vários gerenciadores dele para bancos diferentes. Por exemplo, na configuração acima foi utilizada `ebean.default` que é a configuração padrão do Ebean. Poderíamos ter criado outra entrada, por exemplo `ebean.outrobanco`, apontando para outro pacote caso existisse a necessidade de utilizar dois bancos na mesma aplicação. Na aplicação ficaria assim:

```
1 EbeanServer server = Ebean.getServer("outrobanco");
2 server.save(obj);
```

Para quem tem conhecimento de JPA, é similar a **Peristence Unit**.

Configurando o driver e um pouco de SBT

Quase tudo já está configurado, só falta um último detalhe que é adicionar o driver do MySQL na nossa aplicação. O Ebean já vem com o jar do *HSQLDB*, que é um banco muito utilizado para testes de aplicação mas nunca em produção. Para adicionar o jar, será necessário mexer no arquivo do *SBT*, a ferramenta de build mais utilizada no mundo Scala. O objetivo é o mesmo do Maven, Ant, Gradle entre outros. O leitor não precisa ficar preocupado, só vamos mexer no básico do SBT.

Na raiz do projeto existe um arquivo chamado **build.sbt**. Vamos dar uma olhada inicial.

```
1 name := ""agendatech""
2
3 version := "1.0-SNAPSHOT"
4
5 lazy val root = (project in file(".")).enablePlugins(PlayJava)
6
7 scalaVersion := "2.11.1"
8
9 libraryDependencies += Seq(
10   javaJdbc,
11   javaEbean,
12   cache,
13   javaWs
14 )
```

Já existem até algumas configurações como o nome do projeto e algumas bibliotecas padrões nos projetos Play. Além disso tem uma configuração adicional indicando que foi escolhido a versão Java de desenvolvimento. Basicamente é escrito um pouco de código Scala nesse arquivo. Vamos alterar a lista de dependências, adicionando o MySQL.


```
1  name := ""agendatech""
2
3  version := "1.0-SNAPSHOT"
4
5  lazy val root = (project in file(".")).enablePlugins(PlayJava)
6
7  scalaVersion := "2.11.1"
8
9  libraryDependencies ++= Seq(
10     javaJdbc,
11     javaEbean,
12     cache,
13     javaWs,
14     "mysql" % "mysql-connector-java" % "5.1.20"
15 )
```

É importante que os espaços sejam preservados entre as linhas. É um requerimento do SBT. Agora que adicionamos essas informações, abra o console do Play no seu projeto. Se o console já estiver aberto, com o servidor rodando, pressione **CTRL+D** e depois **Enter** para pará-lo. Com o servidor parado, digite **reload** para as informações do arquivo serem recarregadas.

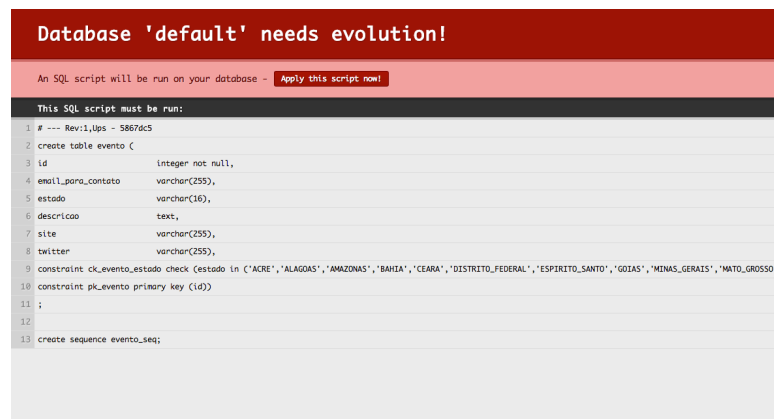
Pronto, agora basta digitar **update** para o Play baixar a nova dependência. Depois da atualização, digite **eclipse** para o seu projeto ser atualizado.

Criando as tabelas com as Evolutions

As classes já estão mapeadas, a configuração do banco já está pronta. Finalmente podemos gravar os dados dos eventos. Os últimos detalhes que faltam são criar o banco de dados em si e criar as tabelas.

Para criar o banco, acesse o terminal do seu computador e abra o mysql. Caso esteja usando Linux ou Mac, o comando seria `mysql -u root`. Lembre-se que caso tenha especificado um password, o mesmo será pedido agora. Vale também alterar o **application.conf** passando o password necessário. Para criar o novo banco, digite `create database agendatech_play;`.

É necessário agora criar as tabelas. Para isso vamos apenas fazer um experimento. Primeiro vamos colocar o servidor para rodar, `~run`. Feito isso, vamos acessar qualquer url da nossa aplicação. Por exemplo `http://localhost:9000`, que é a entrada da app. Quando fizermos essa tentativa, vai ser mostrada para a gente uma tela como a que segue.



Perceba que é uma tela de erro dizendo que precisamos aplicar uma evolução no banco. A ideia é que o Play vá gerando os scripts de atualização das tabelas baseado nos mapeamentos das suas classes. Para comprovar essa teoria vamos dar uma olhada na pasta `conf/evolutions/default`. Ela contém o script que foi gerado baseado nas anotações feitas na classe *Evento*.

```

1  # --- !Ups
2
3  create table evento (
4      id integer auto_increment not null,
5      email_para_contato varchar(255),
6      estado varchar(16),
7      descricao text,
8      site varchar(255),
9      twitter varchar(255),
10     nome varchar(255),
11     constraint ck_evento_estado check (estado in ('ACRE','ALAGOAS','AMAZONAS','BAHIA','CEARA','DISTRITO_FEDERAL','ESPIRITO_SANTO','GOIAS','MINAS_GERAIS','MATO_GROSSO','MATO_GROSSO_SUL','PARA','PARAIBA','PERNAMBUCO','PIAU','PARANA','RIO_DE_JANEIRO','RONDONIA','RIO_Grande_Sul','RORAIMA','SANTA_CATARINA','SERGIPE','SAO_PAULO','TOCANTINS')),
12     constraint pk_evento primary key (id))
13 ;
14
15 # --- !Downs
16
17 SET FOREIGN_KEY_CHECKS=0;
18
19 drop table evento;
20
21 SET FOREIGN_KEY_CHECKS=1;

```

Inclusive foi respeitado a constraint da nossa Enum que representa os estados. Além disso ela tem um especie de comentário especial interpretado pelo Play, que indica qual parte do script é responsável por criar e qual é responsável por destruir coisas no banco de dados. Um outro detalhe importante que é que podemos criar nossas evoluções independentemente das nossas classes. Aqui só juntamos o útil ao agradável. Vamos dedicar um capítulo só para tratar das Evolutions e das possibilidades que são oferecidas.

Para aplicar a evolução, basta clicar no botão “Apply this script now”. Um outro ponto interessante é que, para tratar a ordem das evolutions, o Play cria uma tabela no sistema contendo a data da aplicação da evolução, o script utilizado e tudo mais. O nome da tabela é *play_evolutions*.

Pronto, feito isso já conseguimos cadastrar quantos eventos quisermos.

Listando os eventos e o Redirect

Agora que o cadastro do evento já está sendo feito, precisamos listá-lo. Aqui o processo já é conhecido. Deixe o console do Play rodando para ir acompanhando qualquer problema que possa aparecer e vamos partir para a implementação. Vamos começar pela View. Uma tela com uma listagem bem simples, vamos dar atenção ao layout no fim do livro.

```
1  <html>
2      <body>
3          <ul>
4              @for(evento <- eventos) {
5                  detalhe do evento aqui
6              }
7          </ul>
8      </body>
9  </html>
```

Perceba que já tomei a liberdade de adicionar um código dinâmico na nossa view. Vamos precisar da lista de eventos para que seja possível exibir as informações dos mesmos. Por sinal, o console do Play está reclamando que essa View não compila, o que faz todo sentido pois ainda não existe a variável *eventos*.

O primeiro detalhe a se notar é que para executar qualquer código dinâmico na view, é necessário usar *@* na frente. É similar ao *<%%>* do PHP ou ao *\${}* da JSP no mundo Java tradicional. Precisamos agora descobrir como acessar a variável *eventos*. Vamos só parar para pensar um pouco sobre isso. Já foi discutido que a view do Play é compilada e vira uma classe. Essa classe tem um método estático chamado *render*. Só para lembrar, o mesmo já foi usado.

```
1 public static Result novo() {  
2     return ok(views.html.eventos.novo.render());  
3 }
```

Agora, se o método precisa de informações de fora para rodar, o mais natural é que sejam passados argumentos para ele. É exatamente dessa forma que o Play funciona, quando uma view precisa de objetos do controller, eles são passados via parâmetro.

```
1 @(eventos:List[Evento])  
2 <html>  
3     <body>  
4         <ul>  
5             @for(evento <- eventos) {  
6                 <li>@evento.getNome() - @evento.getSite()</li>  
7             }  
8         </ul>  
9     </body>  
10 </html>
```

Caso parâmetros sejam necessários, eles devem ser a primeira coisa a ser declarada na view. Assim como os pacotes devem ser a primeira coisa a ser declaradas em classes Java. E como já foi visto, se a View não tem parâmetro, basta não fazer nada. Como estamos usando a versão Java do Play, já fica importada na view o pacote *java.util* para que possamos fazer uso das suas classes.

Agora que a tela já foi construída, é necessário criar a action que dá acesso a mesma. Aqui não tem nada de muito novo. Vamos apenas usar o que já estudamos.

```
1 public static Result lista() {  
2     List<Evento> eventos = Ebean.find(Evento.class).findList();  
3     return ok(views.html.eventos.lista.render(eventos));  
4 }
```

E a parte interessante fica por conta da passagem de parâmetros para o método render da classe gerada a partir da View. Nas tecnologias mais comuns, o leitor mais atento vai se lembrar que essa passagem de parâmetros é feita através do binding de uma string com um objeto. Por exemplo: `request.setAttribute("eventos", eventos)`. O que leva a erros de digitação na view ou no próprio controller que são pegos apenas em execução ou até já rodando em produção. Esse é mais um ponto que a compilação nos ajuda muito.

Para finalizar é necessário registrar a rota no arquivo *routes*.

```
1  #outras rotas
2  GET /eventos controllers.EventosController
```

Fazendo redirect

Para finalizar nosso fluxo inicial de cadastro e listagem, precisamos fazer com que após o cadastro de um novo evento o usuário vá para a tela de listagem. Uma primeira abordagem poderia ser a seguinte:

```
1  public static Result cria() {
2      Form<Evento> formFromRequest = eventoForm.bindFromRequest();
3      Evento evento = formFromRequest.get();
4      Ebean.save(evento);
5      return lista();
6  }
```

Como já temos o método que lista, e ele retorna um `Result` para a gente, podemos reaproveitar e jogar o usuário para a tela de listagem. O problema dessa abordagem é que para o navegador, o último request feito foi um post para o endereço `/eventos`. Caso o nosso usuário aperte **F5** para recarregar a página, o navegador vai tentar fazer um segundo post com os últimos dados enviados, o que acarretaria num cadastro duplo.

Aqui é necessário informar ao navegador que queremos fazer um novo request para o endereço da listagem de eventos, o chamado **client redirect**. Dessa maneira, caso o usuário aperte **F5**, ele vai realmente carregar a listagem novamente. Esse é um padrão muito comum na web chamado de **Always redirect after Post**. Vamos ver como ficaria nosso código:

```
1  public static Result cria() throws IOException {
2      Form<Evento> formFromRequest = eventoForm.bindFromRequest();
3      Evento evento = formFromRequest.get();
4      Ebean.save(evento);
5      return redirect(routes.EventosController.lista());
6  }
```

O método *redirect*, também herdado da classe *Controller*, cria um result com o status 303, que significa **See other** e passa para o navegador a próxima URL que ele deve navegar. Caso tenha instalado o *Chrome tools* ou o *Firebug*, isso pode ser facilmente visto.

Conclusão

Nesse capítulo foi coberto a parte inicial de integração do framework de persistência, passando desde o uso do Ebean até a configuração de acesso a banco necessária para realizar as conexões. Além disso

vimos um pouco das evolutions do Play, as quais ainda vão ser bastante utilizadas durante o livro. Por fim chegamos na view de listagem, onde vimos que as telas podem receber parâmetros assim como os métodos que são escritos comumente. Além disso fizemos um redirect depois do cadastro do novo evento, já que não queríamos ninguém apertando F5 e duplicando esses cadastros.

Não pare de ler, no próximo capítulo vamos abordar validação dos dados assim como conversão de tipos de valores que não vem suportados por default no Play.

Convertendo e Validando os dados

O cadastro já está perfeitamente funcional. O problema agora é que é necessário armazenar o período do evento, já que o Agendatech tem o interesse de mostrar os eventos que ainda estão por acontecer, não os que já aconteceram. Para fazer isso, vamos adicionar mais 2 atributos na classe *Evento*.

```
1  @Entity
2  public class Evento {
3      @Id
4      @GeneratedValue
5      private Integer id;
6      private String emailParaContato;
7      @Enumerated(EnumType.STRING)
8      private Estado estado;
9      private String descricao;
10     private String site;
11     private String twitter;
12     private String nome;
13     private Calendar dataDeInicio;
14     private Calendar dataDeFim;
15
16     //getter e setters
```

Agora é necessário adicionar esses 2 campos no formulário de cadastro. Para não perdermos tempo com calendários estilizados e tudo mais, será utilizado o input do tipo *date* do HTML5.

```
1  <html>
2  <body>
3  <form action="@routes.EventosController.cria()" method="POST">
4      <input type="text" id="nome" name="nome" value="" >
5      <input type="text" id="emailParaContato" name="emailParaContato" value="" \
6  >
7      <textarea id="descricao" name="descricao" ></textarea>
8      <select name="estado">
9          @for(estado <- Estado.values) {
10              <option value="@estado.name">@estado.name</option>
11          }
12      </select>
```

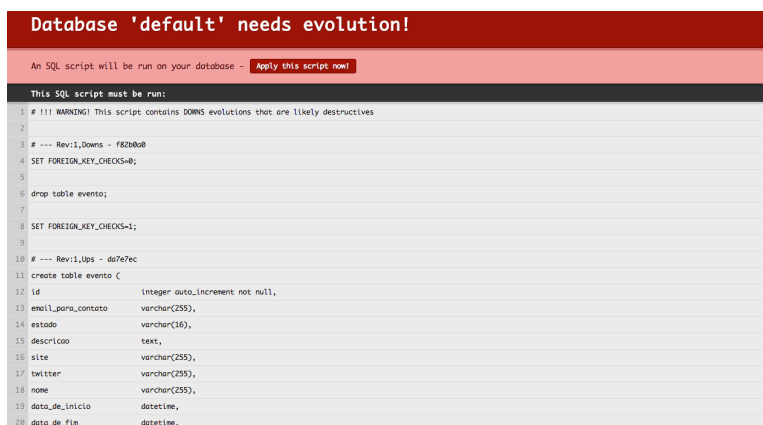
```

13     <input type="text" id="site" name="site" value="" >
14     <input type="date" id="dataDeInicio" name="dataDeInicio" value="" >
15     <input type="date" id="dataDeFim" name="dataDeFim" value="" >
16     <input type="submit" value="Novo evento">
17 </form>
18 </body>
19 </html>

```

Pequeno problema com a Evolution do Ebean

Agora, com as alterações efetuadas, podemos tentar acessar novamente a nossa aplicação. Quando a aplicação é acessada é exibida a tela de erro do Play pedindo para aplicar a evolution gerada por conta da nossa alteração.



Perceba que não é bem isso que queremos. Não queremos dropar a tabela que já existe para poder adicionar novos campos. O desejado é adicionar as novas colunas na tabela já existente. Nesse momento a evolução da tabela está sendo controlada pelo Ebean, que está integrado ao Play. Para contornarmos isso, é necessário apagar os primeiros comentários da evolution.

```

1 # --- Created by Ebean DDL
2 # To stop Ebean DDL generation, remove this comment and start using Evolutions

```

Aqui só um detalhe um pouco obscuro. Lendo o comentário, parece que é para remover apenas a segunda mas na verdade devemos remover as 2 linhas.

Um comentário que na verdade executa código é meio estranho, não se pode negar. Talvez, a melhor solução que o Play poderia ter usado era um mecanismo estilo as migrations do Rails. Poderíamos ter classes que executassem as evoluções com métodos bem definidos de ups e downs.

Agora que o Ebean parou de gerar as evoluções para a gente, devemos criar as mesmas manualmente. Para isso vamos criar o arquivo **2.sql**.


```
1  # --- !Ups
2
3  alter table evento add data_de_inicio date;
4  alter table evento add data_de_fim date;
5
6  # --- !Downs
7
8  alter table evento drop column data_de_inicio;
9  alter table evento drop column data_de_fim;
```

Apenas para lembrar, teremos um capítulo apenas para tratar das evolutions. Por exemplo, e se outro desenvolvedor, em outra máquina, também criar uma evolution com o nome 2.sql, o que deve ser feito? Como aplicar os downs? Essas questões estão separadas em outro capítulo para não perdermos o foco da aplicação agora.

Problema na conversão

Com as novas colunas adicionadas e a classe Evento alterada, podemos tentar cadastrar um novo evento com o período agora!. O problema é que quando tentamos realizar o cadastro recebemos uma exception do tipo *IllegalStateException*.



```
Execution exception
[IllegalStateException: No value]
In /Users/albertoluissoza/ambiente/development/scala/agendatech/app/controllers/EventosController.java at line 30.
27
28 public static Result cria() throws IOException {
29     Form<Evento> formFromRequest = eventoForm.bindFromRequest();
30     Evento evento = formFromRequest.get();
31     Ebean.save(evento);
32     return redirect(routes.EventosController.lista());
33 }
34
35 private static File gravaDestaque() throws IOException {
```

Ela não nos diz muita coisa. Podemos também olhar um pouco no console do Play, mas a stack apresentada não é muito informativa sobre o erro ocorrido. Para tentarmos saber um pouco mais, podemos aprofundar o nível de log do Play. Lembre-se, interpretar bem os erros faz muita diferença na produtividade. Para alterar as propriedades de log do Play, temos que alterar o arquivo `application.conf`.

```
1  # Root logger :
2  logger.root=ERROR
3
4  # Logger used by the framework:
5  logger.play=INFO
6
7  # Logger provided to your application:
8  logger.application=DEBUG
```

Para aumentar o nível de informações de problemas oriundos do core do framework, vamos alterar a entrada **logger.play** para DEBUG ou TRACE. Mesmo assim, o log ainda não nos diz claramente o que está ocorrendo.

Pois bem, o problema que está acontecendo é que o Play, infelizmente, não sabe converter o valor enviado pelos nossos campos de data para um *Calendar*. Ele só consegue trabalhar bem com o tipo *java.util.Date*, que já é sabido que está ultrapassado. Precisamos ensinar ao framework como pegar a String enviada pelo formulário e transformá-la para o tipo que precisamos. É nesse cenário que entram os **Formatters** do Play.

Criando o seu formatter

Primeiramente é necessário criar uma classe que herde de *SimpleFormatter*.

```
1  public class Html5CalendarFormatter extends SimpleFormatter<Calendar> {
2  }
```

Perceba que é feito uso do Generics para que as assinaturas dos métodos que devem ser implementados fiquem mais intuitivas. Os métodos abstratos definidos por esta classe são:

- `Calendar parse(String value, Locale locale)`. Para pegar um String e converter.
- `String print(Calendar value, Locale locale)`. Para pegar um tipo e exibi-lo como String

A implementação deve ficar parecida com o que segue.

```
1  public class Html5CalendarFormatter extends SimpleFormatter<Calendar> {
2
3      private SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
4
5      @Override
6      public Calendar parse(String value, Locale locale) throws ParseException {
7          if (!value.trim().isEmpty()) {
8              Date date = formatter.parse(value);
9              Calendar calendar = Calendar.getInstance();
10             calendar.setTime(date);
11             return calendar;
12         }
13         return null;
14     }
15
16     @Override
17     public String print(Calendar value, Locale locale) {
18         if (value != null) {
19             return formatter.format(value.getTime());
20         }
21         return "";
22     }
23
24 }
```

Perceba que foi usado um formato pouco comum, para nós brasileiros, no parse da data. Isso porque o input do tipo **date** do HTML5 sempre envia a data nesse formato. Baseado na língua do navegador, é exibida a máscara correta ao usuário, porém, o envio dos dados é sempre feito no formato “yyyy-MM-dd”. Dessa maneira o trabalho do servidor é facilitado, não dependemos do *Locale* para saber o formato correto. Apesar de termos a implementação do método *print*, o mesmo ainda não é importante nesse momento. Não fique pensando muito sobre ele, continue comigo na nossa jornada.

Com o **Formatter** implementado, é necessário registrá-lo no Play, para que o mesmo possa usá-lo durante as conversões. Esse registro só precisa ser feito uma vez, O framework já nos fornece uma classe que possuem listeners que são executados em momentos específicos da nossa aplicação, por exemplo quando ela é iniciada. Essa classe é a *GlobalSettings*.

O trabalho se resume a criar uma classe que herde de *GlobalSettings*.

```
1     import java.util.Calendar;
2     import converters.Html5CalendarFormatter;
3     import play.Application;
4     import play.GlobalSettings;
5     import play.data.format.Formatters;
6
7     public class Global extends GlobalSettings {
8
9         @Override
10        public void onStart(Application app) {
11            Formatters.register(Calendar.class, new Html5CalendarFormatter());
12        }
13    }
```

Perceba que não especificamos nenhum pacote, isso porque ela deve estar no pacote default da sua aplicação. Ainda existem listeners para outros eventos como: erros da aplicação, novos requests, etc. Agora podemos tentar efetuar o cadastro de evento que tudo deve ocorrer sem problemas.

Validando os dados básicos

Um outro ponto muito comum em toda aplicação é com a validação dos dados que estão sendo cadastrados. Até esse momento não está sendo feita nenhum tipo de verificação dos dados que estão entrando no nosso sistema. Uma primeira técnica que pode ser utilizada é a de utilização do próprio HTML5 para realizar essa validação.

```
1     <input type="text" id="nome" name="nome" value="" required="required">
```

Por mais que seja válida essa abordagem, nunca é completamente seguro deixar a validação apenas no lado cliente. O código html é facilmente alterável através das ferramentas providas pelos navegadores. Para melhorar a segurança da validação, vamos adicionar um pouco de validação também do lado servidor.

Visando facilitar essa parte, o Play já possui uma integração pronta com a especificação do mundo Java chamada Bean Validation. Ela já provê um mecanismo de validação baseado em annotations que é muito difundido entre os programadores. Vamos ver uns exemplos de validação do nosso modelo:

```
1      import org.hibernate.validator.constraints.NotBlank;
2      import org.hibernate.validator.constraints.URL;
3
4      import play.data.validation.Constraints.Email;
5      import play.data.validation.Constraints.Required;
6
7      @Entity
8      public class Evento {
9
10         @Id
11         @GeneratedValue
12         private Integer id;
13         @Email
14         private String emailParaContato;
15         @Enumerated(EnumType.STRING)
16         private Estado estado;
17         @Column(columnDefinition = "text")
18         @Required
19         private String descricao;
20         @URL
21         private String site;
22         private String twitter;
23         @Required
24         private String nome;
25
26         //resto da classe
27
28     }
```

Perceba pelos imports que a classe está anotada tanto com annotations do Hibernate Validator, que é a implementação principal da especificação, tanto com annotations do próprio Play, que foram criadas seguindo as regras da Bean Validation. Abaixo segue mais algumas annotations úteis.

- `@Length`: verifica o tamanho de alguma String
- `@NotBlank`: verifica se a string está vazia. Já verifica se está nula ou sem nenhum caracter.
- `@SafeHtml`: verifica se o html passado pelo usuário tem algum trecho perigoso. Comum em sistemas que recebem entradas do usuário através dos editores ricos.

Agora que o modelo está devidamente anotado, é necessário executar a validação no controller da aplicação.

```

1     public static Result cria() throws IOException {
2         Form<Evento> formFromRequest = eventoForm.bindFromRequest();
3         if (formFromRequest.hasErrors()) {
4             return badRequest(views.html.eventos.novo.render());
5         }
6         Evento evento = formFromRequest.get();
7         Ebean.save(evento);
8         return redirect(routes.EventosController.lista());
9     }

```

A classe *Form* já possui o método para verificar se existe erros no nosso formulário. Por sinal, esse mesmo método pega qualquer problema de conversão de dados, que era o que estava acontecendo na seção anterior. Se tivéssemos usado ele, seria possível exibir uma mensagem amigável para o usuário. E caso um erro seja encontrado, o usuário é direcionado para a mesma tela que estava antes. Para continuarmos trabalhando com os status corretos do HTTP, é retornado um 400 indicando que o request tinha problemas.

Um detalhe que necessita de preocupação agora é o estado da tela de cadastro quando um erro de validação acontece. O usuário volta para a tela de cadastro com o status apropriado, porém os dados que foram inseridos previamente pelo usuário não estão sendo exibidos para que ele possa preencher os campos problemáticos. Para fazer isso, é necessário que o objeto do tipo *Form* seja disponibilizado na view, dessa maneira será possível recuperar as informações inseridas pelo usuário assim como exibir as mensagens de erro.

```

1     Form<Evento> formFromRequest = eventoForm.bindFromRequest();
2     if (formFromRequest.hasErrors()) {
3         return badRequest(views.html.eventos.novo.render(formFromRequest));
4     }

```

Nesse momento o console do Play vai acusar um erro de compilação informando que o método *render* não espera nenhum argumento. É necessário adicionar o parâmetro na view para que a classe utilitária seja recompilada.

```

1     @(eventoForm:Form[Evento])
2     <html>
3     <body>
4     <form action="@routes.EventosController.cria()" method="POST">
5         <input type="text" name="nome"
6             value="@eventoForm.apply("nome").value()">
7         <input type="text" name="emailParaContato"
8             value="@eventoForm.apply("emailParaContato").value()">
9     <!--outros campos aparecem aqui-->

```

```

10     <input type="submit" value="Novo evento">
11 </form>
12 </body>
13 </html>

```

O método *apply* retorna um objeto do tipo *Field* e através dele temos acesso ao valor que foi enviado no request anterior. O mesmo *Field* retorna os problemas de validação associados à aquele campo. Por sinal esse é um dos principais motivos de usar-se o form para exibir os valores ao invés de um objeto do tipo *Evento*. Porque se estamos perdendo a possibilidade de pegar erros em tempo de compilação, afinal de contas uma string é passada como parâmetro para o método *apply*, ganhamos um objeto que nos dá acesso a informações específicas de cada campo. Abaixo segue o exemplo de como é possível pegar os erros associados a um determinado campo.

```

1  @(eventoForm:Form[Evento])
2  <html>
3  <body>
4  <form action="@routes.EventosController.cria()" method="POST">
5      <div>
6          <input type="text" id="nome" name="nome"
7              value="@eventoForm.apply("nome").value()">
8          <div class="errors">
9              @for(error <- eventoForm.apply("nome").errors()){
10                  <span style="color:red">@error.message()</span>
11              }
12          </div>
13      </div>
14      <input type="text" name="emailParaContato"
15          value="@eventoForm.apply("emailParaContato").value()">
16      <!--outros campos aparecem aqui-->
17      <input type="submit" value="Novo evento">
18  </form>
19  </body>
20  </html>

```

Html helpers para facilitar construção de formulários

Foi utilizado o método *errors* para recuperar todos os erros relativos ao campo nome. O processo seria repetido para todos os outros campos do formulário. E aí é que mais entra a ajuda de um framework na vida do desenvolvedor. Se esse trabalho é repetido várias vezes, não tem motivo para isso ficar a cargo do programador, deveria ser tratado pelo próprio framework. E o Play nos ajuda fornecendo objetos que já constroem os campos do formulário em associação ao objeto *Form*. Veja o mesmo formulário construído usando os helpers de HTML.

```

1      @(eventoForm:Form[Evento])
2      @import helper._
3      <html>
4          <body>
5              @form(routes.EventosController.cria()) {
6
7                  @inputText(eventoForm("nome"))
8                  @inputText(eventoForm("emailParaContato"))
9                  @textarea(eventoForm("descricao"))
10                 <select name="estado">
11                     @for(estados <- Estado.values) {
12                         <option value="@estado.name">@estado.name</option>
13                     }
14                 </select>
15                 @inputText(eventoForm("site"))
16                 @inputText(eventoForm("twitter"))
17                 <fieldset>
18                     <legend>Período</legend>
19                     @inputDate(eventoForm("dataDeInicio"))
20                     @inputDate(eventoForm("dataDeFim"))
21                 </fieldset>
22                 <input type="submit" value="Novo evento">
23
24             }
25         </body>
26     </html>

```

Tenha em mente que o código acima vai gerar o html do formulário para o cadastro do evento. Temos algumas partes importantes que devem ser analisadas. A primeira é o import estático do helper. A sintaxe com underline é do Scala mas a funcionalidade é a mesma do **static import** do Java. Esse pacote helper possui diversos templates de inputs que podemos utilizar. Na tela de cadastro dos eventos, usamos por exemplo: *inputText*, *textarea* e *inputDate* para gerarem o html necessário para cada um dos inputs. Abaixo segue uma amostra do html gerado pelos helpers do Play.

```

1      <dl class=" " id="nome_field">
2          <dt><label for="nome">nome</label></dt>
3          <dd>
4              <input type="text" id="nome" name="nome" value="" >
5          </dd>
6          <dd class="info">...</dd>
7      </dl>

```

E para o caso de uma tela com erros:


```
1 <dl class=" error" id="nome_field">
2   <dt><label for="nome">nome</label></dt>
3   <dd>
4     <input type="text" id="nome" name="nome" value="" >
5   </dd>
6   <dd class="error">O campo não foi preenchido</dd>
7   <dd class="info">Preenchimento obrigatório</dd>
8 </dl>
```

Não precisamos mais ter que ficar escrevendo código para exibir erro de validação associado a um campo nem tão pouco ficar preenchendo o value. O helper do Play já faz isso tudo para a gente. Em um sistema com muitos formulários isso é muito importante para garantir uma boa velocidade na construção dessas telas. O html gerado é baseado em templates já prontos do Play, os mesmos podem ser alterados para se adequarem ao padrão de layout do sistema. Veremos sobre isso no capítulo exclusivo sobre templates!.

Trocando as mensagens de validação

Nesse exato momento, quando o usuário entra na tela de cadastro de eventos, logo abaixo do campo “nome”, está escrito “Required”. Informando que aquele campo é de preenchimento obrigatório. Existe uma informação parecida para o campo de email e assim por diante. Em geral, vamos ter que alterar essas mensagens, assim como as mensagens de erros de validação, para adequá-las as necessidades do cliente. Para realizar essas alterações de mensagens que são exibidas na nossa aplicação, existe o arquivo **messages**, que deve ser criado na pasta **conf**.

```
1 constraint.required = Preenchimento obrigatóriO
2 error.required = O campo não foi preenchido
```

E de onde será que vem essas chaves? Aqui tem um pouco de Bean Validation misturado com um pouco do próprio Play. As annotations de validação definidas pelo framework, segue os requisitos propostos pela especificação. Vamos ver a annotation *Required*.

```
1    @Target({FIELD})
2    @Retention(RUNTIME)
3    @Constraint(validatedBy = RequiredValidator.class)
4    @play.data.Form.Display(name="constraint.required")
5
6    public static @interface Required {
7        String message() default RequiredValidator.message;
8        Class<?>[] groups() default {};
9        Class<? extends Payload>[] payload() default {};
10 }
```

Entre as diversas informações dessa annotation, vamos focar apenas nas que se referem as mensagens. O resto vamos deixar para a seção de validação customizada. O atributo **message** define a chave do properties que será utilizada quando ocorrer um erro de validação, isso é um requisito da especificação. Entretanto, a Bean Validation define que as chaves das mensagens, sempre devem vir entre {} ou seja, ao invés de ser **error.required**, deveria ser {**error.required**}. A decisão de não usar foi tomada pelo Play!. A annotation *Display* recebe a chave do properties referente a mensagem de alerta do usuário. Por isso que usamos as chaves com aqueles nomes!. Para qualquer customização de mensagens, basta analisar esses dois detalhes das annotations e realizar a alteração no arquivo de mensagens. Ainda veremos mais sobre internacionalização no capítulo dedicado apenas a esse tópico.

Criando uma validação reaproveitável

Uma validação necessária para o cadastro do evento é que a data de início seja a partir do dia de cadastro. Por exemplo, se um usuário estiver realizando o cadastro no dia 20/01/2014, a data de cadastro deve ser a partir desse dia. A anotação que mais se aproxima dessa característica é a *Future*, que já vem por default na BeanValidation.

O problema dessa annotation é que ela só considera válida, as datas que vem após o dia de “hoje” e com isso, o cenário descrito acima não seria aceito. Como não existe uma annotation pronta que sirva para esse cenário, vamos criar a nossa.

O primeiro passo é criar a annotation que vai ser usada para o nosso caso de validação. Como é necessário que a data de hoje seja válida, o nome escolhido foi *FromNow*.

```

1  @Target(value={ElementType.FIELD})
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @play.data.Form.Display(name="constraint.fromNow")
5  public @interface FromNow {
6
7      String message() default "error.frowNow";
8
9      Class<?>[] groups() default { };
10
11     Class<? extends Payload>[] payload() default { };
12 }

```

Ela é bem parecida com a annotation *Required*, que foi mostrada um pouco antes no capítulo. Foi usada a annotation *Display* para definir a chave da mensagem que exibe o alerta para o usuário e tem também o atributo *message*, que possui a chave a ser utilizada para definir a mensagem de erro de validação. Os outros dois atributos, *group* e *payload* são detalhes mais específicos. O primeiro serve para agrupar possíveis validações, por exemplo o administrador pode deixar de cadastrar o email de contato enquanto que os outros usuários são obrigados. O payload serve para adicionar um grau de alerta maior para determinados erros de validação, por exemplo uma senha fraca no cadastro de um usuário.

Agora é necessário criar a classe que contém a regra em si da validação. É uma classe comum que implementa a interface *ConstraintValidator*.

```

1  public class FromNowValidator implements ConstraintValidator<FromNow,Calendar\
2  > {
3
4      private FutureValidatorForCalendar futureValidator =
5          new FutureValidatorForCalendar();
6
7      @Override
8      public boolean isValid(Calendar data,
9          ConstraintValidatorContext constraintValidatorContext) {
10         if(data==null) return true;
11         Calendar hoje = Calendar.getInstance();
12         return mesmoDia(hoje, data)
13             ||
14         futureValidator.isValid(data, constraintValidatorContext);
15     }
16
17     private boolean mesmoDia(Calendar hoje, Calendar data) {
18         return hoje.get(Calendar.DAY_OF_MONTH) == data

```

```

19         .get(Calendar.DAY_OF_MONTH)
20         && hoje.get(Calendar.MONTH) == data.get(Calendar.MONTH)
21         && hoje.get(Calendar.YEAR) == data.get(Calendar.YEAR);
22     }
23
24     @Override
25     public void initialize(FromNow fromNow) {
26
27     }
28 }

```

O método *isValid* contém a regra específica. As validações costumam ser bem específicas. Perceba que se o valor da data está nulo, é considerado que o valor está válido. Se o usuário quer proibir datas nulas, ele deveria usar a annotation *NotNull* em conjunto com a nossa. Para não reescrever a regra que verifica se um dia está após o outro, foi feito uso da classe *FutureValidatorForCalendar*, do próprio Hibernate Validator, reutilizando um código que já foi escrito. O método *initialize* é utilizado para o caso da annotation possuir atributos extras que possam ser usados pelo programador para dar uma informação extra na validação. Um caso padrão, é passar a regex na annotation *Pattern*.

O último ponto que falta é fazer a ligação(binding) entre a annotation e a classe de validação. Para isso que serve a annotation *@Constraint*, colocada em cima da annotation que define a validação customizada.

```

1     @Target(value={ElementType.FIELD})
2     @Retention(RetentionPolicy.RUNTIME)
3     @Documented
4     @Constraint(validatedBy=FromNowValidator.class)
5     @play.data.Form.Display(name="validations.FromNow")
6     public @interface FromNow {
7
8         String message() default "error.frowNow";
9
10        Class<?>[] groups() default { };
11
12        Class<? extends Payload>[] payload() default { };
13
14
15    }

```

Para finalizar, basta adicionar as chaves no arquivo **messages**. Caso um usuário preencha uma data de inicio inválida, a nossa validação vai ser usada e não deixará o processo ser completado. A utilização da anotação é bem simples, basta adicioná-la em cima do atributo *dataDeInicio*.

```
1  @FromNow
2  private Calendar dataDeInicio;
```

Validação específica do modelo

Um último ponto relativo ao processo de validação é a verificação do período que será cadastrado para o evento. A regra é simples, se o evento é de apenas um dia de duração, a data de início é igual ao do fim. Caso dure mais, a data de fim deve vir depois da data de início. Essa é uma outra validação que não existe por padrão e, como fizemos a validação básica de data, poderia ser criado uma annotation customizada. Apenas para vermos um outro mecanismo fornecido pelo Play, vamos seguir por um outro caminho que é o de criar um método de validação dentro da classe.

É permitido que seja criado um método chamado *validate* que retorna uma lista de erros de validação, dentro da sua própria classe. No caso da validação de período seria assim:

```
1  public class Evento {
2      //outros atributos e métodos
3
4      public List<ValidationError> validate() {
5          ArrayList<ValidationError> errors = new ArrayList<ValidationError>();
6          if (dataDeFim == null) {
7              this.dataDeFim = (Calendar) dataDeInicio.clone();
8              return null;
9          }
10         if (!dataDeFim.after(dataDeInicio)) {
11             errors.add(
12                 new ValidationError("dataDeFim", "O fim deve ser após o início"));
13         }
14
15         return errors.isEmpty() ? null : errors;
16     }
17 }
```

Caso a lista esteja preenchida o Play entenderá que existem erros de validação. Apenas para deixar mais claro, caso o retorno seja não nulo (inclusive a lista vazia), o framework considera que existem erros de validação. O código é tranquilo, apenas é verificado a regra do período e caso a data de fim não esteja preenchida, ela é igualada a data de início. O construtor da classe *ValidationError* recebe como argumentos o nome do atributo que está sendo validado e a mensagem. Um detalhe importante é que o método *validate* só é executado se as validações baseadas nas anotações passarem. Uma lembrança importante é que o nome do método deve ser **validate**, é uma convenção imposta pelo Play.

Conclusão

Esse foi um capítulo denso. Talvez seja o momento de dar uma respirada, revisar um pouco o que foi feito até agora para depois seguir adiante. Vimos um pouco mais sobre as *Evolutions*, já que tivemos de alterar a estrutura da tabela para adicionar os campos relativos as datas. Também passamos por um problema de conversão de tipos, que foi resolvido através dos *Formatters* do Play o que nos fez ver um pouco mais sobre como estender as funcionalidades que já vem como default. E por fim passamos por vários problemas de validação, os quais foram resolvidos de todas as formas que o Play oferece para nós, programadores.

Fazendo o upload da imagem do evento

O último passo para deixar o cadastro de evento completo, é o envio de uma imagem que representa o evento, uma foto de destaque. Como realizar upload é uma tarefa bem comum, o Play já vem com um suporte básico para o tratamento deste tipo de requisição.

Vamos começar fazendo o mais simples, que é adicionar um novo campo no nosso formulário de cadastro.

```
1      @(eventoForm:Form[Evento])
2      @import helper._
3      <html>
4          <body>
5              @form(routes.EventosController.cria(), 'enctype -> "multipart/form-data") {\
6
7              <!--Outros campos omitidos-->
8                  @inputFile(eventoForm("destaque"))
9              <input type="submit" value="Novo evento">
10          }
11      </body>
12 </html>
```

Não fizemos nada demais aqui. Usamos o helper para construir um input do tipo *File* e, além disso, passamos um novo parâmetro para o método *form*. A sintaxe fazendo uso do *->* é a maneira do Scala de construir entradas para um *HashMap*. E esse é o último parâmetro do *form*, um Hash com as propriedades que queremos associar ao formulário. Nesse caso estamos dizendo que vamos enviar os dados com o Content-Type **multipart/form-data**, a forma mais comum de submeter arquivos através de formulários na web. Todos os helpers de input do Play podem receber esse Hash como último argumento.

Lidando com upload no Controller

É necessário agora lidar com o arquivo enviado no método do Controller. Primeiramente, é preciso acessar o arquivo enviado.

```
1    RequestBody requestBody = request().body();
2    MultipartFormData body = requestBody.asMultipartFormData();
3    FilePart filePart = body.getFile("destaque");
4    File destaque = filePart.getFile();
```

O *RequestBody* consegue acessar os dados enviados na requisição através de vários formatos. Por exemplo:

- `requestBody.asJson()`
- `requestBody.asXml()`
- `requestBody.asFormUrlEncoded()`

Tudo vai depender de como você enviou os dados. Lembre que aqui estamos lidando com dados vindos de requisições dos navegadores, mas o cliente poderia ser qualquer aplicação.

Agora que o arquivo já foi remontado no lado do servidor, precisamos só gravá-lo em algum lugar. Seguindo a convenção do Play, podemos guardá-los na pasta *public/images* e, dentro dela, criar uma outra pasta chamada **destaques**. O código final seria algo parecido com o que segue:

```
1    RequestBody requestBody = request().body();
2    MultipartFormData body = requestBody.asMultipartFormData();
3    FilePart filePart = body.getFile("destaque");
4    File destaque = filePart.getFile();
5    File destino = new File("public/images/destaques", System.currentTimeMillis()
6                          + "_" + destaque.getFilename());
7    FileUtils.moveFile(destaque, destino);
```

O framework ainda carece de um plugin que torne a gravação do arquivo mais fácil. Um detalhe a mais que poderia ser interessante, é de ter a opção de mandar essas imagens para o S3, serviço de storage fornecido pela Amazon e amplamente usado ao redor do mundo. Quem sabe esse não pode ser nosso plugin no final do livro? Inclusive o Heroku, um dos possíveis locais onde podemos realizar o deploy da nossa aplicação, fornece uma implementação para ele. O mesmo pode ser visto no endereço <https://devcenter.heroku.com/articles/using-amazon-s3-for-file-uploads-with-java-and-play-2>.

O framework Rails já possui plugins que além de gravar a imagem que foi submetida, ainda realiza o tratamento das mesmas. Geralmente esse tratamento é feito com a ferramenta **ImageMagick**, que deve ser instalada localmente na máquina.

Integrando o upload com a gravação do evento

Foi reservado uma parte do capítulo apenas para tratar sobre o upload pois o código é um pouco extenso. Agora chegou a hora de integrarmos os trechos de código vistos acima com o código já existente na nossa aplicação. O primeiro passo é separar o processo de construção e gravação do destaque num método privado. Dessa forma mantemos o mínimo de legibilidade no método *cria*.

```
1 public class EventosController extends Controller {
2     //código existente
3
4     private static File gravaDestaque() throws IOException {
5         RequestBody requestBody = request().body();
6         MultipartFormData body = requestBody.asMultipartFormData();
7         FilePart filePart = body.getFile("destaque");
8         File destaque = filePart.getFile();
9         File destino = arquivoDeDestino(filePart);
10        FileUtils.moveFile(destaque, destino);
11        return destino;
12    }
13
14    private static File arquivoDeDestino(FilePart destaque) {
15        return new File("public/images/destaques", System.currentTimeMillis()
16            + "_" + destaque.getFilename());
17    }
18 }
```

Agora basta invocar esse método durante o processo de criação de um evento.

```
1 public class EventosController extends Controller {
2     Form<Evento> formFromRequest = eventoForm.bindFromRequest();
3     if (formFromRequest.hasErrors()) {
4         return badRequest(views.html.eventos.novo.render(formFromRequest));
5     }
6     File destino = gravaDestaque();
7     Evento evento = formFromRequest.get();
8     evento.setCaminhoImagem(destino.getName());
9     try {
10        Ebean.save(evento);
11    } catch (RuntimeException exception) {
12        destino.delete();
13    }
14    return redirect(routes.EventosController.lista());
15 }
```

Dois detalhes a serem notados. Gravamos no banco apenas o nome da imagem, ao invés do binário em si. A ideia é que possamos trocar isso de maneira fácil e manter essas imagens no S3 da Amazon. A segunda é que, caso tenha algum problema durante a gravação do evento, a imagem associada com o mesmo também é deletada. Esse é um passo importante para não ficarmos com muitas imagens que na verdade não pertencem a nenhum evento.

Para finalizar essa parte, precisamos adicionar o atributo **caminhoImagem** na classe *Evento* e, além disso, criar a nova evolution. Abaixo segue o arquivo da evolution. Como essa é nossa segunda evolution, vamos por enquanto chamá-la de **2.sql**.

```
1      # --- !Ups
2
3      alter table evento add caminho_imagem varchar(255);
4
5      # --- !Downs
6
7      alter table evento drop column caminho_imagem;
```

Como é de praxe, na primeira vez que a aplicação for acessada, o Play vai detectar uma nova evolution e solicitar a permissão para que a mesma possa ser executada.

Conclusão

Nesse capítulo foi visto como realizar uploads usando o Play. O código é um pouco extenso, mas será resolvido com a criação de um plugin no final do livro!. Nesse ponto você já tem bastante informação para começar a desenvolver sua nova aplicação. Não perca tempo, a prática é importantíssima para a assimilação do conteúdo.

Enviando emails de maneira assíncrona

Nesse exato momento, qualquer evento cadastrado no Agendatech está sendo exibido na listagem principal. O problema disso é que se algum cadastro malicioso for feito, por conta da falta de um processo de aprovação, o mesmo vai ser exibido no site, ferindo um pouco a nossa reputação. A ideia agora é adicionar o fluxo de aprovação de eventos.

A primeira coisa a ser feita é adicionar um atributo indicando se o evento está aprovado ou não.

```
1  @Entity
2  public class Evento {
3
4      //outros atributos
5      private boolean aprovado;
6
7      //método de validação mais getters e setters.
8  }
```

É necessário também adicionar mais uma evolution.

```
1  # --- !Ups
2
3  alter table evento add aprovado tinyint default 0
4
5  # --- !Downs
6
7  alter table evento drop column aprovado;
```

Perceba que, de propósito, foi deixado o valor default da coluna aprovado com zero. Inicialmente nenhum evento está aprovado, vamos ter que fazer isso manualmente. Para possibilitar essa aprovação será criada uma tela com a listagem de todos eventos. Separados entre aprovados e não aprovados. Como temos feito durante o livro, começaremos pela tela e depois implementaremos o nosso controller. A mesma ficará no pacote `views.eventos.admin`, dessa forma as telas de administração ficam separadas das telas dos eventos normais.

```

1  @(naoAprovados:List[Evento], aprovados:List[Evento])
2  <html>
3      <body>
4          <h3>Não aprovados</h3>
5          <ul>
6              @for(evento <- naoAprovados) {
7                  <li>@evento.getNome() -
8                      <a href="link para aprovacao">Aprovar</a>
9                  </li>
10             }
11         </ul>
12         <hr/>
13         <h3>Aprovados</h3>
14         <ul>
15             @for(evento <- aprovados) {
16                 <li>@evento.getNome()</li>
17             }
18         </ul>
19     </body>
20 </html>

```

Daqui a pouco vamos adicionar a action de aprovação e a entrada para ela no arquivo de rotas. Por isso deixamos aquele texto no atributo *href* do link. Sem perda de tempo, vamos logo criar a action que carrega todos os eventos cadastrados. Para separar as classe de administração, vamos criá-la no pacote `controllers.admin`.

```

1  public class TodosEventosController extends Controller{
2
3      public static Result todos() {
4          List<Evento> aprovados = Ebean.find(Evento.class).where()
5              .eq("aprovado", true).findList();
6          List<Evento> naoAprovados = Ebean.find(Evento.class).where()
7              .eq("aprovado", false).findList();
8          return ok(views.html.eventos.admin.todos_eventos.render(naoAprovados, aprova\
9 dos));
10     }
11 }

```

Temos deixado o código de acesso a banco misturado no controller, até porque só tínhamos chamado o método que gravava e que listava tudo no banco. Agora que existe uma query com condição, chegou a hora de deixar essa lógica dentro de uma classe especializada.

```
1 public class Eventos {
2
3     public static List<Evento> aprovados(boolean situacao) {
4         return Ebean.find(Evento.class).where().eq("aprovado", situacao)
5             .findList();
6     }
7 }
```

Essa classe implementa um Design Pattern conhecido como DAO. Podemos alterar o código para refletir essa alteração.

```
1 public class TodosEventosController extends Controller{
2
3     public static Result todos() {
4         List<Evento> aprovados = Eventos.aprovados(true);
5         List<Evento> naoAprovados = Eventos.aprovados(false);
6         return ok(views.html.eventos.admin.todos_eventos.render(naoAprovados, aprova\
7 dos));
8     }
9 }
```

O código agora ficou bem mais legível. Manteremos o bom senso sobre quando ou não criar métodos dentro do DAO. A regra básica vai ser que quando tiver realmente uma lógica de consulta, com condições ou outras características, o código será isolado. Caso seja apenas uma invocação para um save ou update, o trecho será mantido no controller mesmo. Caro leitor, lembre que colocar o código no lugar certo é uma tarefa complicada, se discordar dessa abordagem fique a vontade para tentar novas estratégias.

Falta apenas adicionarmos a url de acesso no arquivo de rotas e dessa forma a funcionalidade vai estar liberada para o administrador do Agendatech.

```
1 GET /eventos/admin controllers.admin.TodosEventosController.todos()
```

Aprovando o evento

Na tela de administração deixamos um link, propositalmente sem o href, que deve apontar para a url de aprovação. Para começar a resolver esse problema, vamos criar a action responsável por tratar dessa funcionalidade.

```

1  public class TodosEventosController extends Controller{
2      //action de listagem
3
4      public static Result aprova(Integer id){
5          Evento evento = Ebean.find(Evento.class,id);
6          evento.setAprovado(true);
7          Ebean.update(evento);
8          return redirect(controllers.admin.routes.TodosEventosController.todos());
9      }
10 }

```

A lógica em si não tem nada demais, o detalhe que merece uma atenção é que, pela primeira vez, a action recebe um parâmetro. Afinal de contas precisamos do **id** do evento para que possamos realizar o processo de aprovação. Essa configuração deve ser feita no arquivo **routes**.

```

1  GET /evento/admin/aprova/:id controllers.admin.TodosEventosController.aprova(id\
2  : Integer)

```

Repare que usamos a notação **:variável**. Aqui estamos dizendo que na verdade uma parte da url é composta por um parâmetro. O mesmo nome de variável deve ser usado na hora declarar o parâmetro da action na rota, é dessa forma que o Play faz a ligação(binding) entre o que está sendo passado na url e a posição que deve ser passado para a action.

Com a funcionalidade de aprovação momentaneamente concluída, é necessário realizar a alteração na view de administração de eventos para que o link aponte para o lugar correto.

```

1  @(naoAprovados:List[Evento],aprovados:List[Evento])
2  <html>
3      <body>
4          <h3>Não aprovados</h3>
5          <ul>
6              @for(evento <- naoAprovados) {
7                  <li>@evento.getNome() -
8                      <a
9                          href="@admin.routes.TodosEventosController.aprova(evento.getId())"
10                         >Aprovar</a>
11                  </li>
12              }
13          </ul>
14          <hr/>
15          <h3>Aprovados</h3>
16          <ul>

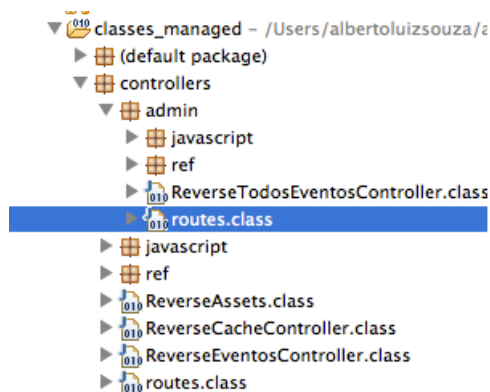
```

```

17         @for(evento <- aprovados) {
18             <li>@evento.getNome()</li>
19         }
20     </ul>
21 </body>
22 </html>

```

Um detalhe importante a ser notado é que tivemos de acessar a classe **routes** colocando o pacote **admin** na frente. Por default, o Play só importa o pacote controllers para a view e, como dentro dele tem a classe gerada automaticamente chamada **routes**, estávamos acessando ela sem maiores problemas. Agora que criamos um pacote novo, o Play gerou uma nova classe **routes** dentro do pacote controllers.**admin** e com isso tivemos que mudar um pouco nossa maneira de acesso. Apenas para clarificação segue a estrutura gerada:



Compondo urls com parâmetros

Voltando um pouco, a composição de url com parâmetros é bastante flexível no Play. Vamos dar uma olhada com mais calma nisso. No exemplo anterior, especificamos que era necessário passar o **id** do evento sendo aprovado como parâmetro.

```

1  GET /evento/admin/aprova/:id controllers.admin.TodosEventosController.aprova(id\
2  :Integer)

```

Caso seja passado um parâmetro não numérico, o Play vai retornar o status **400**, também conhecido como **Bad Request**, indicando que a requisição foi feita com informações inválidas. Ele consegue deduzir isso pois o parâmetro da action recebe um Long. Pensando ainda mais um pouco nessa questão, na verdade a requisição foi feita para uma URL errada!. Como não temos nenhuma pretensão de tratar id's alfanuméricos, podemos adicionar uma regra na nossa rota para o Play retornar **404**, informando que a URL procurada pelo usuário não existe.

```
1 GET /evento/admin/aprova/:id<[0-9]+> controllers.admin.TodosEventosController.a\  
2 prova(id: Integer)
```

É suportado expressões regulares na rotas. Dessa forma o programador tem a liberdade de adicionar algumas regras específicas para dificultar o trabalho de de quem tentar passar argumentos inválidos nas URL's.

Expressões regulares tendem a ser muito complexas se usadas para uma prevenção de parâmetros muito complicada. Lembre de adicionar apenas as básicas, dessa forma ganhamos uma garantia mínima sem comprometer muito a manutenibilidade das rotas.

Conflito entre rotas e valores default

Uma outra funcionalidade que ainda não implementamos é a visualização dos detalhes dos eventos. O código em si, deixaremos mais para frente, mas vamos pensar um pouco em como seria a rota para essa tela.

```
1 GET /eventos/:id controllers.EventosController.visualiza(id: Integer)  
2 GET /eventos/novo controllers.EventosController.novo()
```

Fizemos como de praxe, adicionamos a chamada a possível action do Controller no nosso arquivo. O detalhe que pode nos surpreender aqui é que, caso um usuário acesse o endereço **/eventos/novo**, a primeira rota declarada servirá como resolução para a URL solicitada. O Play vai substituir o parâmetro **:id** pelo palavra **novo**, passado pelo usuário, e vai gerar um erro pois o método espera como argumento um **Integer**.

Esse é um problema clássico em arquivos de configurações de rotas. A primeira encontrada, que casa com a URL passada, é usada para invocar a action. Uma maneira simples de resolver, é inverter a ordem das rotas. O detalhe que o programador tem que ficar atento é que, talvez o sistema contenha tantas rotas, que arrumá-las de uma maneira que a prioridade seja respeitada, vira uma tarefa um tanto complexa.

Uma outra maneira de amenizar o problema das prioridades é com o uso de expressões regulares nas rotas. Com os tipos de valores definidos, o Play vai procurar pela rota cujo parâmetro seja substuível pela regex. Por exemplo:

```
1 GET /eventos/:id<[0-9]+> controllers.EventosController.visualiza(id: Integer)  
2 GET /eventos/novo controllers.EventosController.novo()
```

Estamos garantindo que o id mesclado na URL tem que ser um número. Caso o Play veja o acesso ao endereço **/eventos/novo**, ele automaticamente vai ignorar a primeira rota.



Enviando email

Voltando a última funcionalidade implementada, a aprovação dos eventos, faltou um detalhe importante. Do jeito que está agora, podemos começar a esquecer que os eventos estão sendo cadastrados, pois não é nos avisado sobre nenhum novo cadastro. Para melhorar isso, vamos adicionar um envio de email de alerta a cada novo evento.

Para não perder tempo implementando toda funcionalidade de envio de email, uma coisa bem comum nas aplicações, vamos usar um plugin já pronto. Para encontrar uma lista dos plugins endossados pela Typesafe, empresa que mantém o Play, acesse <https://github.com/typesafehub/play-plugins>. Usaremos o plugin chamado **mailer**, que já possui uma implementação para envio de email com configurações que podem variar em função do ambiente, além de já possuir um **Mailer** de mentira, o que facilita nossos testes.

Para adicionar o plugin ao projeto, basta adicionar as dependências no **build.sbt**, responsável pela configuração dos módulos do nosso projeto.

```
1 libraryDependencies += Seq(  
2     #outras depedencias,  
3     "javax.mail" % "mail" % "1.4.1",  
4     "com.typesafe.play.plugins" %% "play-plugins-mailer" % "2.3.0"  
5 )
```

Agora vamos para a parte mais interessante, que é mandar o email em si. Configurações em arquivos externos estão no final da seção. Na action que cadastra o evento, além de alterar o status do evento, temos que mandar o email para o administrador do sistema.

```
1     public static Result cria() throws IOException {  
2         //codigo de cadastro de evento...  
3         ControladorDeEmails.informaNovo(evento);  
4         return redirect(routes.EventosController.lista());
```

Propositalmente já foi deixado a lógica de enviar em outra classe. Lembre que sua **action**, idealmente, deve apenas invocar os passos responsáveis para executar a sua regra de negócio. A classe **ControladorDeEmails** fica assim:

```
1      public class ControladorDeEmails {  
2  
3  
4          public static void informaNovo(Evento evento){  
5              MailerAPI mail = play.Play.application().plugin(MailerPlugin.class).email();  
6              mail.setSubject("mailer");  
7              mail.addFrom("cadastros@agendatech.com.br");  
8              mail.addRecipient("destino@gmail.com");  
9              Html render = views.html.email.novo.render(evento);  
10             mail.sendHtml(render.body());  
11         }  
12     }
```

A API de envio é auto explicativa, apenas alguns pontos merecem destaque. Sempre que for necessário carregar um plugin pelo Play, será usado um método de fábrica chamado **plugin**, que recebe como parâmetro uma classe que é o ponto de entrada para o uso da extensão. Faremos basicamente isso quando implementarmos o plugin de Upload!.

Um outro ponto muito interessante é que externalizamos o html de envio de email numa view comum. Repare bem nessas linhas de código:

```
1      Html render = views.html.email.novo.render(evento);  
2      mail.sendHtml(render.body());
```

Como nossa view é compilada, podemos acessá-la de qualquer lugar do nosso código. Então fizemos basicamente o que já estamos acostumados a fazer nos controllers, chamamos o método **render** passando o parâmetro que ela espera e depois pegamos o corpo gerado. Muito simples e já suportado pelo framework!. Dessa maneira fica muito fácil criar qualquer template de envio de email para qualquer sistema que será desenvolvido.

Configuração do mailer

É necessário realizar a configuração básica de envio de email, como porta e usuário de acesso. Todas essas configurações devem ficar no arquivo **application.conf**, que concentra as configurações padrões da aplicação.

```
1  #smtp.host (mandatory)
2  #smtp.port (defaults to 25)
3  #smtp.ssl (defaults to no)
4  #smtp.tls (defaults to no)
5  #smtp.user (optional)
6  #smtp.password (optional)
```

Tudo está comentado pois essas configurações vão depender do serviço de email que esteja sendo usado pela aplicação. Uma outra configuração importante, pelo menos para o período de testes da aplicação, é que não queremos disparar emails de verdade, apenas fazer uma simulação. Para isso vamos adicionar mais uma linha de configuração:

```
1  smtp.mock = true
```

Pronto, isso é suficiente para o **mailer** escolher uma implementação de mentira.

Um último ponto a ser configurado é a ordem de carregamento do plugin. A ideia é que você possa escolher por exemplo: se quer ele seja carregado antes ou depois do carregamento das configurações padrões do framework. Uma outra situação podia ser em relação a uma dependência com outro plugin.

Para fazer isso, será criado o arquivo chamado **play.plugins**.

```
1  1500:com.typesafe.plugin.CommonsMailerPlugin
```

Por exemplo, colocando 1500 é garantido que ele seja executado após as configurações do **core** do framework, mas antes do que o **GlobalPlugin**, responsável por chamar a classe **Global**, que foi criada para fazer o registro do **Formatter** de datas. Dentro do próprio framework existe o arquivo de configuração de plugins.

```
1  100:play.api.i18n.DefaultMessagesPlugin
2  100:play.core.RouterCacheLifecycle
3  1000:play.api.libs.concurrent.AkkaPlugin
4  10000:play.api.GlobalPlugin
```

Esse trecho de código pode ser visto no repositório do próprio projeto. <https://github.com/playframework/playfra>

Problemas na execução de código síncrono

Agora que o email pode ser enviado após o cadastro do novo evento, caímos num novo problema. Nesse exato momento, caso nossa aplicação sofra uma chuva de cadastros e o servidor de email esteja um pouco lento por algum motivo, podemos começar a gerar um gargalo. Essa situação pode ser levada para outras aplicações. Imagine um sistema de E-commerce onde o gateway dos pagamentos esteja meio lento. Caso muitas pessoas queiram comprar um produto em promoção, o site pode começar a ficar lento porque esse ponto de integração está engarrafando todas as requisições do sistema.

Na programação, essa situação onde um código fica travado esperando outro terminar, é chamado de processamento síncrono. E muitas vezes é a raiz de vários problemas no nosso código. Geralmente qualquer operação que envolva IO como: acesso a banco de dados, web services e envio de emails são pontos que fazem o sistema entrar em situações de gargalo.

Modelo de trabalho padrão dos servidores

A maioria dos servidores que somos acostumados a usar para desenvolver as aplicações Java, trabalham no modelo de 1 Thread por requisição. Ou seja, se a sua action fizer um processo demorado, outros clientes da aplicação podem começar a sofrer reflexo da sua lentidão.

Servidores como Jetty e Tomcat, já possuem configurações que permitem alterar um pouco esse modelo de trabalho. E a geração mais nova de servidores Web, como o Grizzly e o Undertow, já foram pensados para ser usados por aplicações que não querem ficar travando todo o processamento enquanto uma tarefa pesada está sendo executada. Este é o modelo conhecido como assíncrono.

O grande problema aqui é que a API de Servlet não foi concebida para trabalhar dessa maneira. Então de nada adianta os servidores fornecerem toda a infra estrutura necessária se as API's não podem tirar proveito. Nem mesmo com a adição do suporte a execução de código assíncrono, incluída na especificação de Servlets 3, o problema foi resolvido.

Vários projetos do mundo Java funcionam baseados numa classe chamada **ThreadLocal**, que basicamente é um mapa de Thread(a requisição) para algum objeto. Quando um novo contexto assíncrono é iniciado, novas Threads são geradas e consequentemente esse mapa fica perdido. Os frameworks vem tentando resolver e progressos já podem ser vistos no Spring e no Weld, mas ainda assim não deixa de ser um ajuste e não um modelo que vem pensado desde a criação da API.

O mesmo acontece no mundo Ruby com o servidor **Thin**, por exemplo. A infra estrutura é baseada em eventos assíncronos, mas se as libs utilizadas nos projetos não tirarem proveito disso, de nada vai adiantar essa infra.

Suporte nativo a execução assíncrona

No Play, ao contrário dos casos citados acima, API e infra estrutura estão em perfeita sintonia. Essa é uma das vantagens do Play não usar um servidor que siga a especificação de Servlets. Por um lado

ele perde bastante em padronização mas o ganho em integração é recompensador. O servidor usado pelo framework é baseado no **Netty**(<http://netty.io>), que foi criado com a ideia de suportar vários protocolos e entre eles está o HTTP. Como ele é completamente baseado em eventos, geralmente isso quer dizer que ele funciona de maneira assíncrona, se encaixava perfeitamente no que o Play estava procurando.

Para completar o suporte a execução de código assíncrono dentro de uma action, o Play também é baseado em outro projeto chamado **AKKA**(<http://akka.io/>). Essa biblioteca permite que a comunicação entre as partes do sistema seja feita através de mensagens. Basicamente ao invés de invocar um método em um objeto que você conhece, simplesmente é passado os parâmetros para um método do AKKA e ele os repassa para seu objeto. A vantagem é que o AKKA consegue rodar esse código numa thread separada, controlar o acesso simultâneo, tratar problemas de sincronização, etc. Mais detalhes sobre toda a teoria por trás do AKKA pode ser encontrada no site(<http://akka.io/>) e no excelente curso(<https://www.coursera.org/course/reactive>) disponibilizado pelo Coursera.

Fazendo uma action assíncrona

Dado todo esse contexto, vamos começar a ver como ficaria o nosso código de aprovação sendo executado de maneira assíncrona.

```
1 Promise<Void> enviandoEmail = Promise.promise(new Function0<Void>() {  
2  
3     @Override  
4     public Void apply() throws Throwable {  
5         ControladorDeEmails.informaNovo(evento);  
6         return null;  
7     }  
8 });
```

Não fique assustado, tem muita coisa acontecendo no trecho de código acima. Nesse momento o mais importante é você entender que o objeto do tipo **Promise** representa uma intenção de executar algum código. É muito semelhante ao **java.util.concurrent.Callable** presente na API de futures do Java. Se quisermos ser bastante sinceros, é essa API que está sendo usada lá por debaixo dos panos, a vantagem é que você não precisa lidar com os detalhes de infraestrutura desse código.

O código que deve ser executado é passado através de uma classe anônima, chamada **Function0**, que está no pacote **play.libs.F**. E a tipagem é para dizer que nosso código vai gerar algum retorno, como só queremos mandar o email, vamos usar a classe utilitária **Void**, que nos permite retornar **null** dentro do nosso método.

Agora que temos a intenção criada, precisamos dizer ao Play o queremos fazer quando esse código for executado.

```

1    Promise<Result> result = enviandoEmail.map(new Function<Void, Result>() {
2        @Override
3        public Result apply(Void nada) throws Exception {
4            return redirect(redirect(routes.EventosController.lista()));
5        }
6    });

```

Mais outro trecho um pouco complicado de código. As classes anônimas fazem com que tudo fique um pouco mais difícil. O Play já está sendo preparado para a chegada do Java8 para trocar essas classes anônimas por **Lambdas**.

O método **map** é responsável por pegar o retorno da execução da nossa **Promise**, por isso a primeira tipagem com **Void** e gerar uma nova promise, nesse caso com a intenção de realizar um redirect para listagem dos eventos na pagina inicial da aplicação. Por isso que o segundo parâmetro da tipagem é o **Result**. A classe **Function** também fica no pacote **java.libs.F**.

Agora só precisamos dizer que o retorno da nossa action não é um **Result** e sim uma **Promise**

```

1    public static Promise<Result> cria() throws IOException {
2        //codigo de cadastro de evento...
3
4        Promise<Void> enviandoEmail = Promise.promise(new Function0<Void>() {
5
6            @Override
7            public Void apply() throws Throwable {
8                ControladorDeEmails.informaNovo(evento);
9                return null;
10           }
11       });
12
13       Promise<Result> result = enviandoEmail.map(new Function<Void, Result>() {
14           @Override
15           public Result apply(Void nada) throws Exception {
16               return redirect(routes.EventosController.lista());
17           }
18       });
19       return result;
20   }
21

```

Dessa maneira o Play já sabe que enquanto essa **Promise** não tiver sido cumprida, ele pode atender novas requisições, melhorando a escalabilidade do sistema.

Propositalmente foi deixado o código de acesso ao banco de fora da Promise. Fica a cargo do programador saber quais são os gargalos do sistema. Poderia ter sido feita uma Promise que gravava no banco então poderia ter sido usado o **map** para gerar outra que enviasse o email e por fim uma terceira poderia ser gerada para retornar o Result. A composição vai depender das necessidades do sistema. Divirta-se!.

Conclusão

Nesse capítulo vimos um pouco de como usar um plugin para uma tarefa rotineira, mas o mais importante foi termos tido a ideia de execução de código assíncrono. Essa é uma funcionalidade que deve ser muito explorada, pois os ganhos em escalabilidade são valiosos. Código assíncrono e Cache, assunto do próximo capítulo, são os grandes personagens nos sistemas mais famosos mundo a fora.

Cacheando o resultado das actions

A listagem e a visualização de eventos do Agendatech provavelmente são as funcionalidades, junto com a parte de cadastro, que mais são acessadas no site. A diferença fundamental é que na listagem estamos carregando vários eventos o tempo todo enquanto que no cadastro só exibimos um formulário, basicamente sem informações dinâmicas. No capítulo anterior foi discutido como a realização de tarefas de maneira assíncrona podem ajudar na escalabilidade do sistema. Um outro ponto que pode ajudar tanto na escalabilidade quanto na performance, é não ficar acessando o banco de dados toda hora para recuperar as informações necessárias.

Vamos pensar na listagem. Por mais que muitos eventos sejam cadastrados, esse número talvez chegue a 5 em um mesmo dia, baseado em uma experiência prévia com outros sites de eventos de tecnologia. Partindo desse princípio, a listagem que vai ser exibida para um usuário agora, provavelmente não vai ser diferente da listagem que vai ser exibida para outro usuário daqui a dez minutos. Baseado nessa análise, poderia ser até bem mais profunda dependendo da complexidade da aplicação, podemos manter o html gerado na listagem dos eventos na memória e servi-lo por um tempo determinado para todos os usuários que acessem a aplicação, ao invés de ficar construindo a página o tempo todo. Essa técnica é conhecida como **Cache** e pode ser aplicada para qualquer tipo de leitura de dado, que não precise ser sempre o mais recente, dentro da sua aplicação.

Começando com cache no Play

Usar a API de cache do Play é muito simples. Já existe uma classe chamada **Cache** que nos permite interagir com a implementação de cache usada pelo framework. Para fim de curiosidade é utilizado o **EhCache** como implementação default. O que queremos é guardar o result gerado para a listagem de eventos.

```
1 public static Result lista() {
2     Status result;
3     if (Cache.get("home") == null) {
4         result = ok(views.html.eventos.lista
5                     .render(Eventos.aprovados(true)));
6         int umaHoraEmSegundos = 3600;
7         Cache.set("home", result, umaHoraEmSegundos);
8     }
9
10    return (Result) Cache.get("home");
11 }
```


Como sempre trabalhamos com o objeto do tipo **Result**, o único trabalho é checar se ele já está no Cache e, caso não esteja, basta adicioná-lo. O primeiro parâmetro do método **set** é a chave que você quer associar a entrada que está sendo adicionada no cache. Dessa maneira você pode recuperar esse objeto de qualquer lugar da aplicação. O último indica o tempo que você quer manter a informação cacheada, esse valor é passado em segundos. Da mesma maneira que foi decidido cachear o result inteiro, poderia ter sido decidido cachear apenas o retorno da listagem ou também a visualização do detalhe do evento.

Um ponto que podemos pensar aqui é a relação entre colocar um objeto no cache e a regra de negócio que está sendo executada. Perceba que estamos misturando detalhe de infra estrutura com a lógica específica da aplicação. Ainda tem mais, pode ser que o limite do cache tenha chegado e o mesmo decida colocar parte dos dados em disco para liberar espaço na memória. Fazendo a conexão com o capítulo anterior, qualquer operação de escrita poderia ser realizada de maneira assíncrona e, para nossa sorte, o Play já vem com suporte pronto para adicionar o retorno da nossa action no cache de maneira assíncrona.

Facilitando o trabalho com o @Cache

Indicar ao Play que desejamos que o retorno da action fique no cache é muito simples.

```
1    @Cached(key="home",duration=3600)
2    public static Result lista() {
3        return ok(views.html.eventos.lista.render(Eventos.aprovados(true));
4    }
```

Para fim de curiosidade, vamos olhar um pouco como o Play implementou o código que adiciona o retorno da action no cache.

```
1    public class CachedAction extends Action<Cached> {
2
3        public F.Promise<SimpleResult> call(Context ctx) {
4            try {
5                final String key = configuration.key();
6                final Integer duration = configuration.duration();
7                SimpleResult result = (SimpleResult) Cache.get(key);
8                F.Promise<SimpleResult> promise;
9                if(result == null) {
10                    promise = delegate.call(ctx);
11                    promise.onRedeem(new F.Callback<SimpleResult>() {
12                        @Override
13                            public void invoke(SimpleResult simpleResult) throws Thro\
14    wable {
```

```
15         Cache.set(key, simpleResult, duration);
16     }
17     });
18     } else {
19         promise = F.Promise.pure(result);
20     }
21     return promise;
22 } catch(RuntimeException e) {
23     throw e;
24 } catch(Throwable t) {
25     throw new RuntimeException(t);
26 }
27 }
28
29 }
```

Não é necessário entender o código todo nesse momento, o mais importante é ver que ele coloca o retorno no cache usando a API de **Promise** que vimos no capítulo passado. Dessa forma ele não bloqueia a aplicação enquanto está escrevendo no cache.

Invalidando o cache programaticamente

Em geral, quando é decidido o tempo de expiração do cache, foi feita uma análise básica sobre as atualizações nos dados. Por exemplo no Agendatech não tem muito problema se um novo evento demorar uma hora para aparecer na nossa listagem. Agora, caso o evento que foi cadastrado tivesse pago para aparecer em primeiro, talvez fosse importante já invalidar o cache e recuperar a nova listagem.

É por conta deste tipo de situação que é importante poder invalidar qualquer entrada no cache de maneira programática. Isolaremos essa regra num controller e usaremos a mesma classe **Cache**.

```
1 public class CacheController extends Controller{
2
3     public static Result invalidate(){
4         String key = request().getQueryString("key");
5         Cache.remove(key);
6         return ok();
7     }
8 }
```

Perceba que ela não tem nada demais. Recebe como parâmetro uma chave e remove a entrada associada a ela do cache. O nosso retorno é bem simples também, apenas usa o método `ok` sem parâmetro para indicar que queremos devolver o status 200. Como é de praxe, temos que adicionar a url associada a essa action no arquivo **routes**.

```
1 GET /cache/invalida controllers.CacheController.invalidate()
```

Cuidado com tipo de cache que você deseja

O `@Cache` guarda o retorno da sua action independente do usuário que tenha acessado. Caso você tenha um sistema onde a página cacheada exiba o nome de um usuário logado ou algo parecido, a aplicação pode acabar mostrando a página com o nome de um usuário para outro. Geralmente o uso deste tipo de cache, é mais aplicável a páginas que nada tenham a ver com o usuário.

Uma outra abordagem seria fazer o cache de maneira programática associando um id de usuário na chave, só que desta maneira você poderia acabar com um número de entradas diretamente proporcional ao número de usuários logados. Então pense bem antes de adicionar algum cache de página.

Conclusão

A utilização do cache pode ajudar muito a aplicação tanto para melhorar o tempo de resposta para um usuário(performance) quanto para manter esse tempo de resposta para vários usuários(escalabilidade). Como o uso da API é muito simples, a maior atenção reside em qual dado vai ser eleito para ficar no cache.

Pense sobre a possibilidade dos dados ficarem com uma versão antiga para o cliente, o tempo que eles devem sobreviver no cache e se essas informações são gerais ou específicas para um usuário.

Já pensando nas próximas funcionalidades, você já deve ter percebido que temos algumas funcionalidades que só deveriam estar expostas para administradores do sistema. No próximo capítulo vamos ver como Play nos ajuda quando temos que implementar um sistema de autenticação.

Protegendo o sistema

O Agendatech já possui algumas funcionalidades claramente administrativas, como limpar o cache, aprovar novos eventos. A ideia agora é proteger o acesso a essas funcionalidades através de um sistema de autenticação.

Realizando o login

Vamos começar fazendo o processo de autenticação. Primeiramente, como já estamos fazendo no decorrer do livro, criaremos uma view com o formulário de login do usuário.

```
1      @(usuarioForm:DynamicForm)
2      @import helper._
3      <html>
4          <body>
5
6              @if(usuarioForm.hasGlobalErrors) {
7                  <p class="error">
8                      @usuarioForm.globalError.message
9                  </p>
10             }
11
12             @form(routes.LoginController.loga()) {
13
14                 @inputText(usuarioForm("email"))
15                 @inputPassword(usuarioForm("senha"))
16                 <input type="submit" value="Login">
17
18             }
19         </body>
20     </html>
```

Uma técnica diferente que foi usada para construir este formulário foi que, ao invés de usarmos um `Form[T]`, útil quando temos formulários relacionados com objetos mais complexos, usamos um **DynamicForm**. Esta classe é mais interessante quando o formulário é mais simples, por exemplo este de login. Perceba que o uso dos helpers fica idêntico, a única diferença é que ao invés de ter um objeto do seu modelo por baixo do **Form**, existe um **Map** com as informações a serem exibidas. Uma

outra solução que talvez fosse adotada por programadores mais puristas, seria a da criação de uma classe **Login** só para conter os dados de autenticação. Essa técnica pode ser interessante quando você precisa dos dados desse formulário em mais de um ponto da sua aplicação, ou caso ele realmente ele seja um formulário mais complexo que, por exemplo, talvez necessite de regras de validação.

O form aponta para uma action chamada **loga**, no controller chamado de **LoginController**. Vamos dar uma olhada como ficaria essa implementação.

```
1      public class LoginController extends Controller {
2
3          private static DynamicForm form = Form.form();
4
5          public static Result loga() {
6              Form<Dynamic> requestForm = form.bindFromRequest();
7              String email = requestForm.data().get("email");
8              String senha = requestForm.data().get("senha");
9
10             Usuario usuario = Usuarios.existe(email,
11                 Crypt.sha1(senha));
12
13             if(usuario != nul){
14                 return
15                     redirect(controllers.admin.routes.TodosEventosController.todos());
16             }
17
18             DynamicForm formDeErro = form.fill(requestForm.data());
19             formDeErro.reject("O login ou senha não existem");
20             return forbidden(views.html.login.render(formDeErro));
21         }
22     }
23 }
24 }
```

Tem bastante coisa acontecendo nessa action, vamos começar pelo que já foi comentado. O **DynamicForm** é utilizado para recuperar os valores vindos do request. Perceba que é usado o mesmo método **bindFromRequest** e ele retorna um **Form[Dynamic]** ao invés de um form linkado com uma classe do seu domínio. Através dele conseguimos recuperar os valores passados como parâmetro. Aparentemente nada muito complicado.

A segunda parte já tem a ver com a realização do login dos usuários na aplicação. O interessante é que não tem nada que você não seja capaz de fazer. A classe **Usuarios** concentra a lógica de verificação do usuário no banco, assim como a classe **Eventos**. Além dela precisamos de uma classe que representa o Usuário. Seguindo a convenção, será criada no pacote **models**.

```
1      @Entity
2      public class Usuario {
3
4          @Id
5          private String email;
6          private String senha;
7
8          //apenas para o uso do Ebean
9          @Deprecated
10         public Usuario() {
11             // TODO Auto-generated constructor stub
12         }
13
14         public Usuario(String email, String senha) {
15             this.email = email;
16             this.senha = senha;
17         }
18
19         //getter e setters
20     }
```

Agora a classe **Usuarios**:

```
1      public class Usuarios {
2
3          public static Option<Usuario> existe(String email, String senha) {
4              Usuario usuario = Ebean.find(Usuario.class).where().eq("email", email)
5                  .eq("senha", senha).findUnique();
6              return usuario;
7          }
8
9      }
```

Sem perda de tempo com assuntos já vistos

Temos uma nova classe de modelo, que deve ser mapeada para o banco de dados. Lembre de criar uma evolution para representar esse passo. Assim como temos de configurar no **routes** a rota para a action **loga**.

Um pouco de design

Vamos voltar um pouco ao trecho de código onde verificamos se foi retornado um usuário ou não.

```
1     Usuario usuario = Usuarios.existe(email,
2         Crypt.sha1(senha));
3
4     if(usuario != nul){
5         return redirect(controllers.admin.routes.TodosEventosController.todos());
6     }
```

Esta é uma implementação bem comum quando um método pode retornar ou não um objeto. O problema é que isso não está claro na assinatura do método. O método retorna um usuário e o programador pode ficar tentado a utilizar o objeto direto. Geralmente para descobrir se o retorno pode ser nulo ou não, a pessoa tem que olhar a implementação do método e verificar. Esse tipo de situação é o que nos leva a receber em todo projeto alguns **NullPointerException**. Uma das exceptions mais básicas, mas que não importa o tipo de projeto, sempre acabamos recebendo.

Para tentar diminuir a margem de chance disso acontecer, podemos deixar claro no retorno do método que talvez um usuário não possa ser encontrado.

```
1     public static Option<Usuario> existe(String email, String senha) {
2         Usuario usuario = Ebean.find(Usuario.class).where().eq("email", email)
3             .eq("senha", senha).findUnique();
4         if(usuario==null){
5             return Option.<Usuario>None();
6         }
7         return Option.Some(usuario);
8     }
9 }
```

O retorno do tipo **Option** faz justamente isso, tenta deixar claro que talvez um usuário seja devolvido. A interface **Option** possui duas implementações: **Some** e **None**. Como os próprios nomes sugerem, a primeira indica que alguma coisa vai ser retornada e a segunda, o contrário. Essa classe fica no pacote **play.libs**.

Agora, na action, temos que mudar a maneira como interagimos com o objeto.

```
1     Option<Usuario> talvezUmUsuario = Usuarios.existe(email,
2         Crypt.sha1(senha));
3
4     if(talvezUmUsuario.isDefined()){
5         return redirect(controllers.admin.routes.TodosEventosController.todos());
6     }
```

Perceba que o programador não pode usar os métodos do usuário direto. Sempre vai ter que passar pela **Option**, que está deixando claro que talvez não exista nenhum usuário. Caso algum objeto esteja

definido, basta que seja invocado o método `get`. A invocação do `get` em um `Option` que não tem um objeto definido lançará uma exception de `IllegalStateException`, indicando que não existe valor para ser recuperado.



Mais um tipo de Result

Para finalizar, no último trecho da nossa action, usamos o método `forbidden` para indicar que o usuário não tem acesso ao sistema. Este helper monta um objeto do tipo `Result` com o status `403`. Sempre é interessante retornarmos os status corretos para cada situação, dessa maneira a aplicação cliente pode tomar melhores decisões sobre os retornos das actions.

Restringindo o acesso

Os usuários já estão sendo logados, mas o problema de acesso liberado ainda persiste. Qualquer usuário pode apontar para a url que cuida da aprovação de eventos e realizar essa aprovação sem se autenticar no sistema.

O primeiro passo a ser dado é verificar se existe algum usuário logado antes de liberar o acesso a essa parte da aplicação.

```
1 public static Result todos() {
2     if(usuario logado){
3         List<Evento> aprovados = Eventos.aprovados(true);
4         List<Evento> naoAprovados = Eventos.aprovados(false);
5         return ok(views.html.eventos.admin.todos_eventos.render(naoAprovados,
6             aprovados));
7     }
8     return redirect(routes.LoginController.form());
9 }
```

Como saber se o usuário está logado ou não? O leitor mais experiente já deve ter pensado, basta colocar o usuário na sessão da aplicação. E é exatamente essa a resposta!.

Sessão é uma maneira de mantermos os dados referentes a um determinado usuário por diversos requests desse mesmo usuário. Em geral é criado uma entrada no cookie do navegador do cliente com um id para aquele usuário. No servidor é criada uma estrutura que associa esse id com algumas informações específicas do usuário.

Quando o usuário efetua o login, temos que adicionar a informação pertinente a ele na sessão. Dessa maneira podemos fazer a verificação no nosso `if`.

```
1    public static Result loga() {
2        //mesmo codigo de antes
3
4        if(talvezUmUsuario.isDefined()){
5            session().put("email", talvezUmUsuario.get().getEmail());
6            return redirect(controllers.admin.routes.TodosEventosController.todos());
7        }
8
9        //continuacao
10       return forbidden(views.html.login.render(formDeErro));
11
12    }
```

Através do método `session` temos acesso ao objeto do tipo `Session`, que nos permite adicionar informações na sessão do usuário.

Agora no código que mostra os eventos que ainda estão pendentes para aprovação, podemos usar essa mesma sessão.

```
1    public static Result todos() {
2        if(session().containsKey("email")){
3            List<Evento> aprovados = Eventos.aprovados(true);
4            List<Evento> naoAprovados = Eventos.aprovados(false);
5            return ok(views.html.eventos.admin.todos_eventos.render(naoAprovados,
6                                                                    aprovados));
7        }
8        return redirect(routes.LoginController.form());
9    }
```

Um pouco mais sobre sessions no Play

O programador mais experiente pode estar se perguntando porque não foi deixado o objeto completo do tipo `Usuario` na sessão. O motivo é uma restrição imposta pelo próprio framework.

A sessão web é uma maneira que foi encontrada pelos desenvolvedores de manter estado do cliente no lado do servidor. O problema principal disso é que, se a aplicação precisa ser rodada em vários servidores (escalabilidade horizontal), esse estado mantido em um servidor começa a ser o gargalo para a escalabilidade. A aplicação vai ter que encontrar uma maneira de compartilhar os dados mantidos numa máquina com os outros servidores que fazem parte do balanceamento de carga.

Como a proposta do Play é ser o mais sem estado possível(stateless), eles só permitem que sejam adicionados strings na session. Toda informação lá adicionada é escrita no cookie do cliente e passada em toda requisição.

Cookie: treeForm_tree=hi+treeForm:tree:applications; JSESSIONID=08ab0743aad8e472ae7a1e565f8; PLAY_SESSION="6e238338cab51e8ee9a38a3dc0b249c6d434a-email=alots.ssa@gmail.com"

Perceba que a chave email faz parte do valor da entrada **PLAY_SESSION**. Dessa forma sua aplicação pode escalar para quantos servidores ela quiser. Como o dado é passado em toda requisição, não importa em qual servidor a requisição caia.

Caso você ainda queira manter estado no servidor, é aconselhado que seja usada a API de cache mostrada no capítulo anterior. A implementação padrão dela é baseada na biblioteca EhCache e a mesma já possui implementação para rodar de modo distribuído. Outra implementação que pode ser escolhida é o **Infinispan**, projeto da Red Hat que é usado para cache no servidor de aplicação Wildfly.

Só precisa tomar um cuidado. Na API de cache, o próprio programador fica responsável por cuidar das associações de chaves. Por exemplo:

```
1 Cache.set("usuario",objetoUsuario);
```

Esse código resultará em apenas um usuário no cache, afinal de contas o cache é global no Play. Lembre de sempre associar um identificador na chave para diferenciar um usuário do outro. Geralmente o id do banco já vai servir.

Compondo actions para evitar duplicação de código

A abordagem de verificar se existe o email na sessão de fato resolve o nosso problema. O detalhe é que temos de fazer essa verificação para o controle do cache, listagem de eventos aprovados, aprovação de eventos, etc. Da maneira atual vamos ter que espalhar esse **if** em todo lugar.

Para esse tipo de situação, onde um código deve ser executado sempre antes de uma action, o Play disponibiliza o mecanismo de **Action Composition**.

```
1 public class AuthenticatedAction extends Action.Simple{
2     @Override
3     public Promise<Result> call(Context ctx) throws Throwable {
4         //codigo antes
5
6         delegate.call(ctx);
7
8         //codigo depois
9     }
10 }
```

Herdando da classe interna **Action.Simple** você sobrescreve o método `call` e consegue executar um código antes de uma action específica. A variável **delegate** representa a action que deverá ser invocada de fato. Dessa maneira conseguimos fazer o controle de usuário logado apenas nesse trecho de código.

```

1      public class AuthenticatedAction extends Action.Simple{
2
3          @Override
4          public Promise<Result> call(Context ctx) throws Throwable {
5              String email = ctx.session().get("email");
6              if(email!=null){
7                  return delegate.call(ctx);
8              }
9              //é só para envelopar numa promise mesmo. Não quer dizer que é um processo \
10 assincrono. Podia ser.
11              ctx.flash().put("nao_logado","Para continuar precisa estar logado");
12              return F.Promise.pure(redirect(routes.LoginController.form()));
13          }
14      }
15
16  }
```

O método `flash` deverá ser usado quando você quer adicionar um objeto que deva durar até o próximo request. É tipicamente usado para adicionar mensagens que devem aparecer depois de um redirect. O exemplo acima é um caso típico.

O próximo passo é informar ao Play quais actions devem ser interceptadas pelo nosso controlador de usuários autenticados. Essa configuração deve ser feita através da annotation **@With**.

```

1      @With(AuthenticatedAction.class)
2      public static Result todos() {
3          List<Evento> aprovados = Eventos.aprovados(true);
4          List<Evento> naoAprovados = Eventos.aprovados(false);
5          return ok(views.html.eventos.admin.todos_eventos.render(naoAprovados,
6                          aprovados));
7      }
```

Podemos colocar em cima de métodos específicos ou direto na classe quando a intenção é interceptar todos os métodos. Agora basta configurar cada ponto da aplicação que necessita de autenticação para ser acessado.

Plugin de autenticação do Play

Como autenticação é algo muito comum nas aplicações, o Play já vem com um plugin para esse caso. O funcionamento é bem parecido com o implementado por nós na seção anterior. Ao invés de usar a annotation `@With`, vamos utilizar uma outra annotation chamada `@Security.Authenticated`.

```
1  @Security.Authenticated(PlayAuthenticatedSecured.class)
2  public class TodosEventosController extends Controller {
3      //implementacao aqui
4  }
```

Apenas para fins de curiosidade vamos dar uma olhada nessa annotation:

```
1  @With(AuthenticatedAction.class)
2  @Target({ElementType.TYPE, ElementType.METHOD})
3  @Retention(RetentionPolicy.RUNTIME)
4
5  public @interface Authenticated {
6      Class<? extends Authenticator> value() default Authenticator.class;
7  }
```

Repare que ela usa o `@With` para indicar que essa annotation deve ser lida como uma configuração de composição de actions. Você também pode usar esse recurso para criar suas próprias composições customizadas.

Como agora vamos usar a action já implementada pelo Play, devemos apenas criar uma classe que implementa a maneira de pegar a informação da sessão e para onde o usuário deve ser redirecionado, caso ele não esteja logado.

```
1  public class PlayAuthenticatedSecured extends Security.Authenticator{
2
3      @Override
4      public String getUsername(Context ctx) {
5          return ctx.session().get("email");
6      }
7
8      @Override
9      public Result onUnauthorized(Context ctx) {
10         ctx.flash().put("nao_logado", "Para continuar precisa estar logado");
11         return redirect(routes.LoginController.form());
12     }
13
14 }
```

Perceba que essa classe foi justamente o parâmetro passado para a annotation `@Security.Authenticated`. Por sinal esse é o mecanismo mais indicado para adicionar controle de autenticação a seu sistema. Aproveite o que já está pronto e foque ao máximo na sua própria lógica.

Conclusão

Nesse capítulo foi visto como implementar uma funcionalidade muito comum em aplicações web, o controle de usuários autenticados. Além disso vimos a composição de actions, que é uma maneira elegante de não duplicarmos código entre todas as actions do nosso sistema.

Não pare agora, o próximo capítulo vai mostrar como a mesma action pode servir formatos diferentes de resposta. É um capítulo curto e muito útil!. Gaste mais 10 minutos e ganhe mais um pouco de conhecimento.

Servindo formatos distintos

Até esse momento o Agendatech só retorna HTML como resposta. Nada mais natural já que nosso cliente é um navegador. Só que agora existe a necessidade que os eventos também sejam exibidos em um aplicativo mobile.

Como estamos falando de outra aplicação consumindo nossas informações, deve ser definido um formato de dado a ser trocado pelas partes. Atualmente, um dos formatos preferidos pelas aplicações é o JSON. O Play já vem com um suporte muito bom, vamos dar uma olhada.

```
1 public static Result lista() {  
2     List<Evento> aprovados = Eventos.aprovados(true);  
3     JsonNode json = Json.toJson(aprovados);  
4     return ok(json);  
5 }
```

Qual formato servir?

Só que agora a aplicação caiu num problema porque se o request for feito pelo navegador vamos querer devolver HTML, mas se for feito pelo cliente mobile, o formato retornado deverá ser JSON. Para resolver essa questão vamos deixar que o cliente nos informe qual formato ele prefere, e nossa aplicação vai retornar o resultado ou informar que não suporta aquele formato.

Sempre que um request HTTP é feito, o cliente pode passar um cabeçalho chamado **Accept**, informando qual o formato de resposta que ele prefere. Por exemplo quando o seu navegador faz o request, o Accept é parecido com o que segue:

```
1 text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;
```

Perceba que o formato preferido, nesse caso é **text/html**. Com esse header em mãos, nossa action pode decidir qual resposta gerar. Esse mecanismo é conhecido como **Content Negotiation**, muito utilizado em integração de sistemas baseados em **REST**. O código em si não é nada muito complicado.

```
1    @Cached(key = "home", duration = 3600)
2    public static Result lista() {
3        List<Evento> aprovados = Eventos.aprovados(true);
4        if(request().accepts("text/html")){
5            return ok(views.html.eventos.lista.render(aprovados));
6        }
7        if (request().accepts("application/json")){
8            JsonNode json = Json.toJson(aprovados);
9            return ok(json);
10       }
11       return status(NOT_ACCEPTABLE);
12   }
```

Simplesmente fazemos a checagem dos headers e damos o retorno esperado pelo cliente. Caso o formato não seja suportado, retornamos o status **406**, que indica formato não aceitável.

Para fazer o teste, você pode instalar o plugin do **Google Chrome** chamado Postman ou usar o programa chamado CURL. Usando o último, podemos ir no console e fazer as seguintes chamadas: `curl --header "Accept: text/html" http://localhost:9000` e depois `curl --header "Accept: application/json" http://localhost:9000`. A action deve dar uma resposta diferente para tipo de chamada.

Problema gerado pelo @Cached

Caso você tenha feito o teste acima, deve ter percebido que depois do primeiro request o resultado é sempre o mesmo, independente do formato pedido. Esse problema acontece por conta da utilização da anotação `@Cached`. Depois do primeiro request, o resultado é mantido no cache e a partir do segundo nossa lógica não é mais executada.

Essa é uma limitação que não pode ser superada no momento. Uma possível solução é criar duas actions, uma para servir JSON e outra para servir HTML. Uma outra sugestão, é voltar a utilizar a API de cache manualmente.

```
1    public static Result lista() {
2        List<Evento> aprovados = Eventos.aprovados(true);
3        MediaRange media = request().acceptedTypes().get(0);
4        String mediaType = media.toString();
5        if(Cache.get("home_"+mediaType)!=null){
6            Status status = (Status) Cache.get("home_"+mediaType);
7            return status;
8        }
9        if(request().accepts("text/html")){
```

```
10         Status status = ok(views.html.eventos.lista.render(aprovados));
11         Cache.set("home_text/html",status);
12         return status;
13     }
14     if (request().accepts("application/json")){
15         Status status = ok(Json.toJson(aprovados));
16         Cache.set("home_application/json",status);
17         return status;
18     }
19     return status(NOT_ACCEPTABLE);
20 }
```

Perceba que foram adicionado mais um *if* no código para tratar a chave do cache em função do formato solicitado. Lembre que a anotação `@Cached` é apenas uma customização do Play para você usar a Composite Action dele chamada **CachedAction**. Caso ache que esse suporte é muito importante para você, cria a sua composição e use na sua aplicação. Ou melhor ainda, tente fazer uma contribuição para o framework!.

Conclusão

Nesse capítulo curto vimos como servir formatos diferentes dentro dos nossos controllers. Além disso demos uma rápida passada pelo problema gerado com o uso da annotation `@Cached`. Vá em frente e sirva quantos formatos quiser, o Play já tem suporte nativo também a XML caso essa seja sua necessidade!.

Testes automatizados

Nosso sistema já está com as funcionalidades principais implementadas. Cadastramos e listamos os eventos que foram aprovados. Podemos ainda adicionar mais coisas, como só listar os eventos que estão para ocorrer, filtros por estado, associar os eventos a grupos de usuários, etc. Você já é capaz de implementar tudo isso. Um detalhe muito importante, que não foi tratado até esse momento, é a parte de testes da nossa aplicação. Não queremos ficar rodando tudo manualmente para saber se as coisas estão funcionando.

Existem algumas categorias de teste, geralmente divididos entre testes de unidade, de integração e de aceitação. A primeira categoria não vai ser abordada por este livro pois o Play não tem nada de novo para oferecer nessa área. Você continuará usando O **JUnit** da mesma maneira que já deve ter lido em vários lugares. Para uma base maior sobre a teoria relacionada a cada tipo de teste, aconselho o livro **TDD no Mundo Real**, do especialista e ótimo autor Maurício Aniche.

Teste de integração nos Dao's

Na classe *Eventos* temos um método que deve retornar os eventos de acordo com o status da aprovação. Mesmo sendo uma regra simples, ela já envolve a construção de uma query no seu banco e se você pensar um pouco, essa é uma regra muito importante para a aplicação. Nós não podemos mostrar eventos que não foram aprovados.

Para começar vamos criar um novo caso de teste focado em testar as regras escritas na classe *Eventos*.

```
1 public class EventosTest {  
2     @Test  
3     public void deverialistarApenasOsAprovados() {  
4     }  
5 }
```

O nome do método já indica o que estamos querendo verificar. Agora vamos montar nosso cenário de teste, algo muito comum. Precisamos criar alguns eventos, aprovados e por aprovar, para verificar se o método realmente está funcionando direito.

```

1  public class EventosTest {
2      @Test
3      public void deveriaListarApenasOsAprovados() {
4          Evento evento = novoEvento(true, "aprovado@gmail.com");
5          Evento evento2 = novoEvento(false, "naoAprovado@gmail.com");
6          Ebean.save(Lists.newArrayList(evento, evento2));
7
8      }
9  }

```

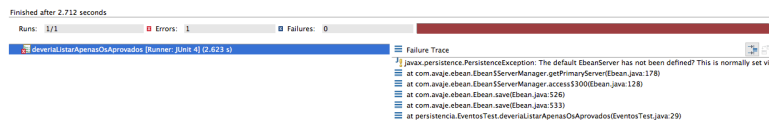
Com o cenário pronto, devemos verificar se, após a invocação do método aprovados, o resultado é o que esperamos. Para isso vamos usar os métodos utilitários do próprio JUnit para nos ajudar.

```

1  public class EventosTest {
2      @Test
3      public void deveriaListarApenasOsAprovados() {
4          Evento evento = novoEvento(true, "aprovado@gmail.com");
5          Evento evento2 = novoEvento(false, "naoAprovado@gmail.com");
6          Ebean.save(Lists.newArrayList(evento, evento2));
7
8          List<Evento> aprovados = Eventos.aprovados(true);
9          Assert.assertEquals(1, aprovados.size());
10         Assert.assertEquals("aprovado@gmail.com", aprovados.get(0)
11             .getEmailParaContato());
12     }
13 }

```

Perceba que é importante verificar se o evento retornado realmente é o que foi aprovado. Caso você teste apenas o tamanho da lista, pode achar que seu teste está passando, mas na verdade está sendo retornado o evento não aprovado. Isso é o que chamamos de **falso positivo**. Quando o retorno é uma lista, geralmente é importante fazer este tipo de verificação. Agora precisamos rodar nosso teste, vamos tentar fazer isso por dentro do próprio Eclipse.



A exception demonstra que o Ebean não foi configurado apropriadamente. Faltou a configuração relativa as propriedades do banco de dados, que já estão lá no arquivo **application.conf**. Esse é um problema que você enfrentará em qualquer framework que tente te ajudar em todas em todas as

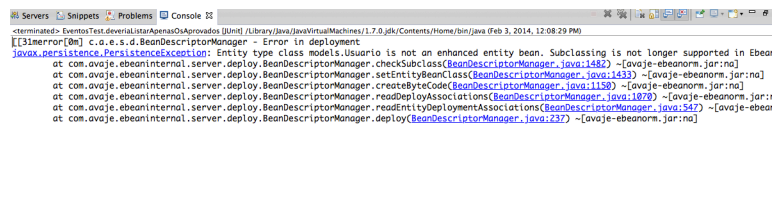
partes da aplicação, do controller à camada de persistência, os chamados **Full Stack Frameworks**. Como toda essa integração é feita pelo framework, precisamos dele rodando para conseguir fazer nossos testes.

A parte interessante é que o Play, assim como no Rails, também se preocupou com isso e já fornece uma maneira de você rodar seu teste dentro de uma simulação da sua aplicação real!.

```
1      import static play.test.Helpers.*;
2
3      public class EventosTest {
4          @Test
5          public void deveriaListarApenasOsAprovados() {
6              running(fakeApplication(), new Runnable() {
7
8                  public void run() {
9                      Evento evento = novoEvento(true, "test@gmail.com");
10                     Evento evento2 = novoEvento(false, "test2@gmail.com");
11                     Ebean.save(Lists.newArrayList(evento, evento2));
12                     List<Evento> aprovados = Eventos.aprovados(true);
13                     Assert.assertEquals(1, aprovados.size());
14                     Assert.assertEquals("test@gmail.com", aprovados.get(0)
15                                     .getEmailParaContato());
16                 }
17             });
18         }
19     }
20 }
```

Agora pode dar uma respirada, porque aconteceu muita coisa. O método **running** é o ponto de entrada para rodarmos um código como se estivéssemos dentro da nossa aplicação de verdade. Para completar, precisamos passar como parâmetro uma instância da classe **FakeApplication**, ela contém a infra necessária para rodar nosso bloco de código. Por exemplo, acesso aos arquivos de configurações. O último parâmetro, a instância de **Runnable**, é apenas para passarmos o código que deve ser executado depois que a aplicação é iniciada. Com a chegada do Java8, esse código deve ficar mais simples. Por fim, foi feito o *static import* dos métodos da classe **Helpers**, que sempre deve ser utilizada durante este tipo de teste.

Com essa alteração vamos tentar rodar os testes, mais uma vez por dentro do Eclipse.



Agora recebemos mais uma exception, essa um pouco mais estranha. Vamos dar uma olhada com um pouco mais de calma.

- 1 javax.persistence.PersistenceException: Entity type class models.Usuario is not
- 2 an enhanced entity bean. Subclassing is not longer supported in Ebean

Para não perdermos muito tempo nisso. Essa exception quer dizer que o bytecode da nossa classe deve ser alterado para que o Ebean possa funcionar. O objetivo disso é que ele consiga ter controle sobre os objetos carregados, por exemplo não carregar duas vezes o objeto com o mesmo id. Antigamente ele fazia isso criando uma classe filha em tempo de execução, agora isso é feito através de instrumentação no momento do carregamento das classes.

A melhor maneira de resolver este problema é rodar os testes através do console do Play!. Ele já possui um comando chamado **test** que roda nossos testes e já instrumenta as nossas classes. Para testar, você pode entrar no console e digitar test.

```
[info] pers
[info] + de
[info]
```

Este tipo de teste, que nos obriga a falar diretamente com alguma parte de infraestrutura da nossa aplicação, é chamado de **Testes de Integração**. Eles fazem bastante sentido para partes do sistema que não podem ser simuladas. Por exemplo, trabalhar com um banco de dados de mentira pode dar a falsa impressão que nossa aplicação está funcionando corretamente.

Trocando as configurações do ambiente de testes

Nesse exato momento estamos rodando nossos testes contra o banco de dados usado durante o desenvolvimento. Com isso, deixamos dados provenientes dos testes no banco que usamos para desenvolver, o que pode causar alguma confusão.

Nesse caso, para não misturar os objetivos, o ideal seria rodar os testes em outro banco de dados, específico para os testes. Quando criamos o objeto do tipo *FakeApplication*, podemos passar para ele um *Map* com as configurações que queremos usar.

```
1    HashMap<String, String> confs = new HashMap<String, String>();
2    confs.put("db.default.url", "jdbc:mysql://localhost/agendatech_play_test");
3    running(fakeApplication(confs), new Runnable() {
4        //codigo de testes aqui
5    }
```

Dessa maneira podemos sobrescrever quaisquer configurações que existam no arquivo `application.conf`.

Uma outra maneira, que este autor acha mais indicada, é usar um arquivo diferente para a configuração dos testes. A ideia é criar um arquivo chamado `test.conf` na pasta de configurações e carregá-lo durante a execução dos testes. Para fazer isso, basta que uma alteração seja feita no arquivo `build.sbt`.

```
1    //restante do arquivo
2
3    javaOptions in Test += "-Dconfig.file=conf/test.conf"
```

Agora, sempre que os testes forem rodados pelo console do play, o arquivo de testes vai ser utilizado. Dessa maneira você tem um lugar, já padrão no framework, onde pode ver as configurações para testes.

O problema de limpar o banco entre os testes

Agora vamos adicionar mais um teste para nossa lógica de listar os eventos. Queremos saber se ele consegue listar os eventos que ainda não foram aprovados.

```
1    //outro teste que pega apenas os aprovados.
2
3    @Test
4        public void deverialistarApenasOsNaoAprovados() {
5        running(fakeApplication(), new Runnable() {
6
7            @Override
8            public void run() {
9                popula();
10               List<Evento> naoAprovados = Eventos.aprovados(false);
11               Assert.assertEquals(1, naoAprovados.size());
12               Assert.assertEquals("test2@gmail.com", naoAprovados.get(0)
13                                   .getEmailParaContato());
14           }
15       });
16   }
```

Depois de rodar os testes com o comando `test`, percebemos que o segundo teste falhou.

```
1  [error] Test persistencia.EventosTest.deveriaListarApenasOsNaoAprovados failed:\n2  expected: <1> but was: <2>
```

A mensagem é clara, no primeiro assert é indicado que esperamos apenas um evento aprovado, mas foi retornado dois eventos. O motivo para isso é que o Play não limpa as tabelas do banco de dados durante a execução dos testes. No nosso caso, como os dois testes inserem dados através do método `popula`, acaba que um está influenciando no outro. Uma solução sugerida pela documentação é a de usar um banco de dados em memória, mas isso pode levar a complicações por conta de alguma query específica do seu banco de dados.

Uma outra solução que pode ser usada é a de limparmos os dados entre os testes.

```
1  public void dropDb() {\n2\n3      String serverName = "default";\n4\n5      EbeanServer server = Ebean.getServer(serverName);\n6\n7      ServerConfig config = new ServerConfig();\n8\n9      DdlGenerator ddl = new DdlGenerator();\n10     ddl.setup((SpiEbeanServer) server, new MySQLPlatform(), config);\n11\n12     // Drop\n13     ddl.runScript(false, ddl.generateDropDdl());\n14\n15     SqlUpdate dropEvolutions = Ebean.createSqlUpdate("drop table play_evolutions\n16 ");\n17     Ebean.execute(dropEvolutions);\n18\n19 }
```

O método `runScript` do Ebean é o responsável por dropar as tabelas existentes. O problema é que se pararmos por aí, a tabela `play_evolutions`, que controla a execução das mesmas, fica populada como se o Play já tivesse rodado as evolutions. Para contornar isso, é necessário também dropar essa tabela. Dessa forma, o framework sempre vai criar tudo para a gente. Em um sistema muito grande, essa abordagem pode ficar um pouco lenta e aí, talvez valha a pena você entrar com um banco em memória, por exemplo o H2.

Agora basta que esse método seja invocado ao fim de cada teste que for executado.

```
1      @Test
2      public void deveriaListarApenasOsAprovados() {
3          running(fakeApplication(), new Runnable() {
4
5              @Override
6              public void run() {
7                  popula();
8                  List<Evento> aprovados = Eventos.aprovados(true);
9                  Assert.assertEquals(1, aprovados.size());
10                 Assert.assertEquals("test@gmail.com", aprovados.get(0)
11                                     .getEmailParaContato());
12                 dropDb();
13             }
14         });
15     }
16
17     @Test
18     public void deveriaListarApenasOsNaoAprovados() {
19         running(fakeApplication(), new Runnable() {
20
21             @Override
22             public void run() {
23                 popula();
24                 List<Evento> naoAprovados = Eventos.aprovados(false);
25                 Assert.assertEquals(1, naoAprovados.size());
26                 Assert.assertEquals("test2@gmail.com", naoAprovados.get(0)
27                                     .getEmailParaContato());
28                 dropDb();
29             }
30         });
31     }
```

Normalmente o `popula` e o `dropDb` ficariam nos métodos anotados com `@Before` e `@After` do JUnit. O problema é que precisamos rodar esses códigos dentro do contexto de execução da *FakeApplication*. Por isso que eles estão dentro do `run`.

Como esses códigos vão ser bem comuns dentro dos testes que você venha a fazer, esses métodos utilitários serão extraídos para uma classe base de testes.

```
1      public class TestHelper {
2
3          public void dropDb() {
4
5              String serverName = "default";
6
7              EbeanServer server = Ebean.getServer(serverName);
8
9              ServerConfig config = new ServerConfig();
10
11              DdlGenerator ddl = new DdlGenerator();
12              ddl.setup((SpiEbeanServer) server, new MySqlPlatform(), config);
13
14              // Drop
15              ddl.runScript(false, ddl.generateDropDdl());
16
17              SqlUpdate dropEvolutions =
18                  Ebean.createSqlUpdate("drop table play_evolution");
19              Ebean.execute(dropEvolutions);
20
21          }
22
23
24      /**
25       * Caso alguém queira popular o banco para uma situação.
26       */
27      protected void popula() {
28      }
29
30      protected void executaFakeApplication(final Runnable runnable) {
31          running(fakeApplication(), new Runnable() {
32
33              @Override
34              public void run() {
35                  try{
36                      popula();
37                      runnable.run();
38                  }
39                  finally {
40                      dropDb();
41                  }
42              }
43          })
44      }
```



```
43         });
44     }
45
46     protected Evento novoEvento(boolean aprovado, String emailContato) {
47         Evento evento = new Evento();
48         Calendar hoje = Calendar.getInstance();
49         Calendar amanha = Calendar.getInstance();
50         amanha.add(Calendar.DAY_OF_WEEK, 2);
51         evento.setDataDeInicio(hoje);
52         evento.setDataDeFim(amanha);
53         evento.setDescricao("descricao teste");
54         evento.setAprovado(aprovado);
55         evento.setNome("evento de teste");
56         evento.setEmailParaContato(emailContato);
57         return evento;
58     }
59 }
```

E nos nossos testes, basta que seja invocado o método `executaFakeApplication` com a implementação de *Runnable* específica da situação.

Testes de integração para os controllers

Uma outra parte interessante de testarmos é se alguns fluxos dos nossos controllers estão sendo respeitados. Por exemplo: um usuário só pode acessar a administração depois de estar logado, um evento deveria aparecer na home depois de aprovado, etc.

O Play já fornece vários métodos utilitários para deixar essa tarefa bem mais simples. Vamos começar verificando que um usuário não pode acessar a administração caso não esteja logado.

```
1 public class AutenticacaoTest extends TestHelper {
2     @Override
3     protected void popula() {
4         novoEvento = novoEvento(false, "contato@gmail.com");
5         Ebean.save(novoEvento);
6         Ebean.save(new Usuario("test@gmail.com", Crypt.sha1("123456")));
7     }
8
9     @Test
10    public void naoDeveriaAcessarAAdministracaoSemEstarLogado() {
11        executaFakeApplication(new Runnable() {
12
```

```

13         @Override
14         public void run() {
15             Result result = callAction(
16                 controllers.admin.routes.ref.TodosEventosController
17                     .todos(), fakeRequest());
18             Assert.assertEquals("/login", redirectLocation(result));
19
20         }
21     });
22 }
23 }

```

Aqui já estamos usando a nossa classe base para ajudar nos testes. O helper **callAction** faz o trabalho sujo de invocar nossa action todos como se fosse um request normal de um usuário. O segundo argumento, um request de mentira, é opcional e só deveria ser usado caso você quisesse passar parâmetros para sua action. Perceba que o retorno é um objeto do tipo **Result**, o mesmo que você retorna por uma action normal.

O nosso teste precisa verificar se você foi redirecionado para a url de login, já que o usuário não está autenticado. Para checar isso existe o método **redirectLocation**, que recebe um result e retorna a string do redirect. Lembre que todos esses métodos vieram do import estático da classe **Helpers**. Para verificar se após o login o usuário está sendo direcionado para a administração, o fluxo seria o mesmo. Fica como desafio para o caro leitor.

Um outro teste útil é o de checar o fluxo de aprovação de um evento. Seria algo parecido com o que segue abaixo:

```

1 public void depoisDeAprovadoOEventoDeveriaAparecerNaHome() {
2     executaFakeApplication(new Runnable() {
3
4         @Override
5         public void run() {
6             FakeRequest requestDeLogin = fakeRequest()
7                 .withFormUrlEncodedBody(
8                     ImmutableMap.of("email",
9                                     "test@gmail.com", "senha",
10                                     "123456"));
11
12             callAction(
13                 controllers.routes.ref.LoginController.loga(),
14                 requestDeLogin);
15
16             callAction(

```

```

17         controllers.admin.routes.ref.TodosEventosController
18             .aprova(novoEvento.getId()),
19         fakeRequest());
20     Result result = callAction(
21         controllers.routes.ref.EventosController.lista(),
22         fakeRequest());
23     }
24 });
25 }

```

O problema dessa parte do teste é que não vamos conseguir acessar a action de aprovação sem estar logado. Uma possível solução seria realizar o login antes.

```

1  @Test
2  public void depoisDeAprovadoOEventoDeveriaAparecerNaHome() {
3      executaFakeApplication(new Runnable() {
4
5          @Override
6          public void run() {
7              FakeRequest requestDeLogin = fakeRequest()
8                  .withFormUrlEncodedBody(
9                      ImmutableMap.of("email",
10                                     "test@gmail.com", "senha",
11                                     "123456"));
12
13              callAction(
14                  controllers.routes.ref.LoginController.loga(),
15                  requestDeLogin);
16
17              callAction(
18                  controllers.admin.routes.ref.TodosEventosController
19                      .aprova(novoEvento.getId()),
20                  fakeRequest());
21              Result result = callAction(
22                  controllers.routes.ref.EventosController.lista(),
23                  fakeRequest());
24          }
25      });
26 }

```

Só que aí caímos num outro problema. O Play não vai compartilhar os dados da primeira requisição com a segunda, a session criada pela primeira requisição não vai ser enxergada pela segunda. Para

resolver isso é necessário que passemos como parâmetro um request com os dados que seriam enviados via cookie e guardados na session.

```
1  @Test
2  public void depoisDeAprovadoOEventoDeveriaAparecerNaHome() {
3      executaFakeApplication(new Runnable() {
4
5          @Override
6          public void run() {
7
8              FakeRequest requestDeLogin = fakeRequest().withSession(
9                  "email", "test@gmail.com");
10
11              callAction(
12                  controllers.admin.routes.ref.TodosEventosController
13                      .aprova(novoEvento.getId()),
14                  requestDeLogin);
15              Result result = callAction(
16                  controllers.routes.ref.EventosController.lista(),
17                  fakeRequest());
18
19              String html = contentAsString(result);
20              Assert.assertTrue(html.contains(novoEvento.getNome()));
21          }
22      });
23 }
```

Como nossa checagem é baseado na chave **email** na session, tudo vai funcionar!. Para finalizar, o método **contentAsString** recebe o result e nos retorna a string gerada pelo nossa view!. Com isso em mãos fica fácil saber se o evento aprovado apareceu na listagem.

Testes de aceitação

Os testes de integração em geral vão compor uma fatia grande do conjunto de testes do seu projeto, junto com os testes de unidade. Mas ainda temos uma última situação que talvez eles não sejam os ideais. Por exemplo, podemos criar um teste para a action que cadastra um novo evento, no estilo dos que criamos acima, o problema é que essa é uma parte tão importante do nosso sistema que talvez fosse mais válido testar como se fosse um usuário real que navega pelas páginas, preenche os campos e clica no botão.

Para cobrir estes cenários, que em geral podem ser alguns poucos mas muito importantes, podemos fazer uso dos **Testes de aceitação**. O Play já vem com uma integração fina com a biblioteca chamada **WebDriver** que nos permite criar testes automatizados para navegação de telas.

```
1  @Test
2  public void deveriaCadastrarUmNovoEvento() {
3      running(testServer(9001, fakeApplication()), HTMLUNIT,
4              new F.Callback<TestBrowser>() {
5
6                  @Override
7                  public void invoke(TestBrowser navegador) throws Throwable {
8                      navegador.goTo("http://localhost:9001/eventos/novo");
9                      navegador.fill("input[name='nome']").with("Conexão Java");
10                     navegador.fill("input[name='emailParaContato']").with(
11                         "contato@caelum.com.br");
12                 }
13             });
14 }
15 }
```

Perceba que ao contrário dos outros testes, fomos obrigados a subir um servidor real numa porta. O parâmetro **HTMLUNIT** indica que vamos utilizar um navegador sem interface gráfica, mas poderíamos ter passado por exemplo **FIREFOX**, dessa forma seria aberto uma instância do Firefox para rodar nossos testes.

O último parâmetro é um callback, assim como era a instância de **Runnable** passada para os testes de integração. Como era necessário receber um parâmetro que representa justamente o navegador sendo utilizado, o Play teve que criar uma nova interface. A classe **TestBrowser** tem métodos bastante intuitivos e o código fica quase que auto explicativo. Primeiro navegamos para um endereço e depois conseguimos preencher os inputs necessários.

Agora vamos dar uma olhada no código completo do teste já utilizando a nossa classe base de testes.

```
1  public class CadastroEventoTest extends TestHelper{
2
3      @Test
4      public void deveriaCadastrarUmNovoEvento() {
5          F.Callback<TestBrowser> fluxoDoUsuario = new F.Callback<TestBrowser>() {
6
7              @Override
8              public void invoke(TestBrowser navegador) throws Throwable {
9                  navegador.goTo("http://localhost:9001/eventos/novo");
10
11                 navegador.fill("input[name='nome']")
12                     .with("Conexão Java");
13                 navegador.fill("input[name='emailParaContato']").with(
14                     "contato@caelum.com.br");
15             }
16         };
17     }
18 }
```

```
15
16         navegador.fill("textarea[name='descricao']").with(
17             "Evento para quem está começando no java");
18         navegador.fill("select[name='estado']")
19             .with("SAO_PAULO");
20
21         navegador.fill("input[name='site']").with(
22             "http://www.conexaojava.com.br/");
23         navegador.fill("input[name='twitter']").with(
24             "@conexaojava");
25
26         navegador.fill("input[name='dataDeInicio']").with(
27             "2014-04-13");
28
29         navegador.fill("input[name='dataDeFim']").with(
30             "2014-04-16");
31
32         navegador.submit("input[type='submit']");
33
34         Assert.assertEquals(
35             "evento cadastrado com sucesso, aguarde a moderação",
36             navegador.$(".msg_sucesso").getText());
37     }
38 };
39 executaAceitacao(fluxoDoUsuario);
40 }
41 }
```

Perceba que já é um código de testes bem maior por conta da manipulação dos campos do formulário. A parte importante é que se for feita uma atualização nessa view e por exemplo o nome de um input for modificado, nosso teste vai quebrar e teremos um feedback rápido.

Os testes de aceitação por serem mais demorados devem ser criados depois de bastante análise. Veja quais as telas que realmente são importantes para o sistema e que não podem quebrar!.

Conclusão

Os testes formam uma das partes mais importantes da aplicação. Através deles temos feedback rápido sobre quaisquer alterações no código e conseguimos reagir rapidamente a qualquer problema. Os helpers oferecidos pelo Play tornam a tarefa do desenvolvedor muito mais tranquila, permitindo que a escrita dos mesmos seja feita de maneira clara e objetiva.

Layout e reaproveitamento de views

Um ponto esquecido pela nossa aplicação, pelo menos até agora, foi a aparência da mesma. Por mais que não seja o tipo de coisa mais apreciado pelos programadores, é necessário dar uma atenção para a construção do HTML respeitando o design criado.

Para facilitar a nossa vida, vamos importar tudo relativo a essa parte do Agendatech que já está rodando. Como estamos recriando a aplicação usando outra tecnologia, faz sentido reaproveitar tudo de CSS, Javascript e HTML já criado no site atual.

Organizando os recursos estáticos

O leitor mais atento deve ter percebido que no arquivo **routes** existe uma rota associada com um controller chamado **Assets**.

```
1 GET /assets/*file controllers.Assets.at(path="/public", file)
```

Como não foi a gente que criou essa classe, a única opção que sobra é que ela foi entregue pronta pelo Play. Esta classe é responsável por carregar qualquer arquivo estático requisitado pelo cliente. Note que qualquer url começada em **assets** vai ser tratada pela action **at** deste controller. Agora a primeira coisa que vamos fazer é alterar nossa tela de listagem para respeitar o layout do Agendatech original importando os css's.

```
1 <head>
2   <title>Agendatech - Eventos de tecnologia</title>
3   <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/st\
4 yle.css")"/>
5   <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/fo\
6 rm.css")"/>
7   <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/co\
8 ntato.css")"/>
9   <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/jq\
10 uery-ui.css")"/>
11 </head>
```

Nada que já não estamos acostumados. Usamos a rota reversa para fazer referência a action e passamos os parâmetros necessários. No arquivo de rotas está definido um valor default para o parâmetro **path**, por sinal isso é uma feature da linguagem Scala. Esse valor default quer dizer

que todos esses recursos estáticos, também chamados de **Assets**, estão na pasta *public*. O segundo argumento indica a localização e aí temos que passar o resto do caminho. A convenção do Play indica o seguinte:

- `public/javascripts` para arquivos javascript.
- `public/stylesheets` para arquivos css
- `public/images` para as imagens da aplicação

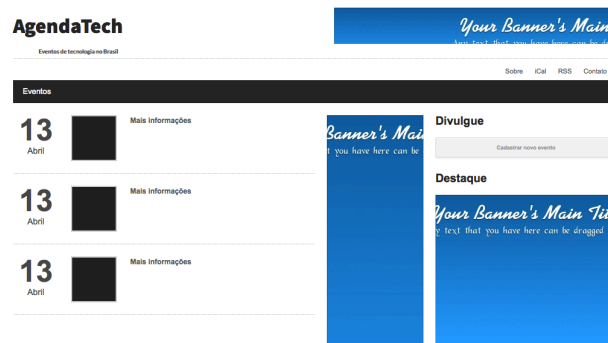
A classe **controllers.Assets** é capaz de fazer muito mais do que apenas achar o asset e servir para a gente. Essa mesma classe pode controlar recursos como:

- Cache
- Gzip
- Minificação

Por isso é altamente indicado que você sempre a use.

Criando o template do sistema

Vamos agora implementar toda a parte de layout. Primeiro vamos ver qual é a cara que queremos.



Tudo isso é obra de HTML + CSS que de longe não é a parte mais importante aqui do livro. Vamos focar em como tirar proveito das views do Play para montar essas telas.

Um detalhe importante que devemos ter em mente é que não importa qual é a tela do *agendatech*, o topo, footer e a parte do banner lateral vão ser as mesmas.

Assim como fazemos para o código Java em si, aplicaremos o princípio do DRY(Don't repeat yourself) nas nossas views. Vamos isolar toda essa parte que se repete num único arquivo de view, por exemplo `main.scala.html`


```

1      <html>
2
3      <!-- HEAD INFO -->
4      <!-- ===== \
5  ===== -->
6      <head>
7          <title>Agendatech - Eventos de tecnologia</title>
8          <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheet\
9  s/style.css")"/>
10         <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheet\
11  s/form.css")"/>
12         <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheet\
13  s/contato.css")"/>
14         <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheet\
15  s/jquery-ui.css")"/>
16         <link href='http://fonts.googleapis.com/css?family=Source+Sans+Pro:400,90\
17  0' rel='stylesheet' type='text/css'/>
18         <link href='http://fonts.googleapis.com/css?family=Lato:100,400' rel='sty\
19  lesheet' type='text/css'/>
20     </head>
21
22     <!-- BODY -->
23     <!-- ===== \
24  ===== -->
25     <body>
26
27         <div class="boxed">
28
29             <!-- HEADER -->
30             <!-- ===== -->
31             <div id="header">
32                 @header()
33             </div>
34
35             <!-- MAIN -->
36             <!-- ===== -->
37             <div id="content">
38
39                 <div style="clear: both"></div>
40                 <div class="container">
41
42                     <!-- MENU -->

```

```

43         <!-- ===== \
44 ===== -->
45         @menu()
46
47         <div class="left-container">
48             <!-- E QUAL CONTEÚDO VAI AQUI NO MEIO?-->
49         </div>
50
51     </div>
52
53     <!-- RIGHT CONTENT -->
54     @right_content()
55
56     <div style="clear: both"></div>
57
58 </div>
59
60 </div>
61
62 <div style="clear: both"></div>
63
64 <div id="footer">
65
66     <div class="container">
67         <a href="/">
68             <p class="logo_footer">Agenda<span>tech</span></p>
69         </a>
70
71         <p><a href="@routes.EventosController.lista">Home</a> |
72         <p class="copy">Copyright © 2010. All Rights Reserved.</p>
73     </div>
74 </div>
75
76     <script src="@routes.Assets.at("javascripts/jquery-1.4.4.min.js")" type="te\
77 xt/javascript"></script>
78
79 </div>
80 </body>
81 </html>

```

O arquivo é meio longo mas não tem quase nada que você já não tenha visto. Caso esteja com alguma dúvida nesse momento, não fique com pressa, a mesma será tirada em poucos minutos. Um detalhe interessante é o comentário com letras maiúsculas no meio do arquivo indicando que ali deve ir o

conteúdo específico de uma página. A ideia é que as páginas específicas, por exemplo a listagem de eventos, possa passar apenas essa parte do html para essa nossa view que é o **template** de páginas do sistema.

E é justamente isso que vamos fazer! Caso você tenha esquecido, nossas views são compiladas para classes com o método `render` e os mesmos podem receber parâmetros!. Basicamente o que queremos é que seja passado o html que deve ser colocado bem ali naquela espaço. Com isso em mente, nossa view ficaria da seguinte forma:

```
1      @(content: Html)
2      ...
3      ...
4      <div class="left-container">
5          @content
6      </div>
7      ...
8      ...
9      </html>
```

Na view de listagem de eventos podemos simplesmente invocar o método `render` da *main* passando o conteúdo que deve ser colocado ali.

```
1      @(eventos:List[Evento])
2
3      @main {
4          @if(flash().get("sucesso")!=null){
5              <span class="msg_sucesso">@flash().get("sucesso")</span>
6          }
7          <ul>
8              @for(evento <- eventos) {
9                  <div class="evento clearfix" id="evento_@evento.getId()">
10                     <div class="data">
11                         @if(evento.taRolando()){
12                             <p>@Calendar.getInstance().dayAsNumber</p>
13                         }else {
14                             <p>@evento.getDataDeInicio().dayAsNumber</p>
15                         }
16                         <small>@evento.getDataDeInicio().monthAsString</small>
17                     </div><!-- fim class="data" -->
18
19                     @if(evento.temLogo()){
20                         <div class="logo_img clearfix">
21                             
23     </div>
24 }else{
25     <div class="logo clearfix"> </div>
26 }
27
28 <div class="info">
29     <p id="evento"><a href="">Mais informações</a></p>
30     <small></small><br/>
31
32     <p class="tags">
33     </p>
34     <br/>
35 </div>
36
37 </div>
38
39     }
40 </ul>
41
42 }
```

Perceba que não foi necessário nem invocar o método `render`. Isso acontece porque a classe gerada possui também um método chamado `apply` que, na linguagem Scala, pode ser invocado sem a necessidade de utilização do nome. Fazer `@main.apply {...}` e `@main {}` dá exatamente no mesmo.

Melhorando a legibilidade com includes

Dentro do loop que monta os eventos para nossa listagem temos um código um pouco extenso. Além do mais esse mesmo trecho de HTML deverá servir também para montar a página de detalhe do evento, busca por eventos, etc...

Para não ficar repetindo esse código podemos pegar apenas esse trecho e isolar numa outra view.

```

1      @(evento:Evento)
2      <div class="evento clearfix" id="evento_@evento.getId()">
3          <div class="data">
4              @if(evento.taRolando()){
5                  <p>@Calendar.getInstance().get(Calendar.DAY_OF_MONTH)</p>
6              }else {
7                  <p>@evento.getDataDeInicio().get(Calendar.DAY_OF_MONTH)</p>
8              }
9              <small>@evento.getDataDeInicio().getDisplayDate(Calendar.MONTH,Calendar.L\
10 ONG,new Locale("pt","BR"))</small>
11          </div><!-- fim class="data" -->
12
13          @if(evento.temLogo()){
14              <div class="logo_img clearfix">
15                  
17              </div>
18          }else{
19              <div class="logo clearfix"> </div>
20          }
21
22          <div class="info">
23              <p id="evento"><a href="">Mais informações</a></p>
24              <small></small><br/>
25
26              <p class="tags">
27                  </p>
28              <br/>
29          </div>
30
31      </div>

```

De novo estamos trabalhando com o fato da view gerar uma classe com um método que pode receber parâmetros. Definimos que precisamos de um evento como argumento e agora qualquer tela da nossa aplicação pode invocar essa view para ter esse trecho de código gerado. Vamos ver como ficaria a listagem.

```
1      @(eventos:List[Evento])
2
3      @main {
4          @if(flash().get("sucesso")!=null){
5              <span class="msg_sucesso">@flash().get("sucesso")</span>
6          }
7          <ul>
8              @for(evento <- eventos) {
9                  @detalhe_lista(evento)
10             }
11         </ul>
12
13     }
```

Muito mais fácil de ler! Esse recurso de isolar um trecho de HTML e reaproveitá-lo é chamado de **include** no desenvolvimento web. E com as views poderosas do Play fica muito fácil de utilizar. Use a abuse disso para deixar suas telas mais as legíveis possíveis. Talvez você tenha percebido que no template principal do sistema foram usadas diversas includes para organização do menu, header, etc.

Um pouco mais de Scala para as views

Um último ponto interessante da nossa listagem de eventos é a exibição do dia e mês que o evento vai ocorrer. Só para ficar mais claro vamos dar uma olhada no layout desejado.



Deve ser exibido apenas o número do dia e apenas o nome do mês que o mesmo vai acontecer. Um exemplo de código para isso seria o seguinte:

```
1      Calendar data = Calendar.getInstance();
2      data.get(Calendar.DAY_OF_MONTH);
3      data.getDisplayName(Calendar.MONTH,Calendar.LONG,new Locale("pt","BR"));
```

O problema deste tipo de lógica na página é que rapidamente isso pode ficar um pouco mais complicado. Por exemplo se tivermos que verificar a preferência de língua enviada pelo navegador e escrever o nome do mês em função disso. O ideal é deixarmos essas lógicas de view isoladas em alguma outra classe.

O caminho natural seria escrever uma classe utilitária tipo a que segue.

```
1 public class FormatadorDeDatas {
2     public static int dayAsNumber(Calendar calendar) {
3         return calendar.get(Calendar.DAY_OF_MONTH);
4     }
5
6     public static int monthAsString(Calendar calendar) {
7         return calendar.getDisplayName(Calendar.MONTH, Calendar.LONG, new Locale(\
8 "pt", "BR"));
9     }
10 }
```

E agora na listagem essa classe poderia ser utilizada.

```
1 @if(evento.taRolando()){
2     <p>@FormatadorDeDatas.dayAsNumber(Calendar.getInstance())</p>
3 }else {
4     <p>@FormatadorDeDatas.dayAsNumber(evento.getDataDeInicio())</p>
5 }
6 <small>@FormatadorDeDatas.monthAsString(evento.getDataDeInicio())</small>
```

Já está bem melhor. Quem vem do mundo do desenvolvimento padrão em Java, utilizando JSP's como view, faz este tipo de trabalho com as taglibs do pacote JSTL.

Indo além agora, utilizando um pouco de código Scala, podemos deixar o código acima ainda mais simples. Imagine que ao invés de ficarmos utilizando essa classe o tempo todo, pudéssemos simular que o próprio Calendar tenha esses métodos. Algo assim:

```
1 @import infra.FormatadorDeDatas._
2
3 @if(evento.taRolando()){
4     <p>@Calendar.getInstance().dayAsNumber</p>
5 }else {
6     <p>@evento.getDataDeInicio().dayAsNumber</p>
7 }
8 <small>@evento.getDataDeInicio().monthAsString</small>
```

Pois bem, podemos fazer isso através de um pequeno trecho de código Scala.

```
1      object FormatadorDeDatas {
2
3          implicit def calendar2PimpedCalendar(calendar: Calendar) = new {
4              def dayAsNumber = calendar.get(Calendar.DAY_OF_MONTH)
5
6              //como o mês começa em zero posso usar ele como indice do array
7              def monthAsString = calendar.getDisplayName(Calendar.MONTH,Calendar.LONG,\
8 new Locale("pt", "BR"))
9
10         }
11     }
```

O método marcado com a palavra chave **implicit** faz com que o compilador do Scala identifique qualquer chamada aos métodos `dayAsNumber` e `monthAsString` num objeto do tipo *Calendar* e já retorne o novo objeto que possui estes dois métodos. Dessa maneira conseguimos simular adição de novos métodos em classes já existentes. Para o compilador não tem que buscar isso o tempo todo, devemos importar a classe nos locais que desejamos essa funcionalidade. Por isso a linha com o `import` na nossa view.

Caso você queira esse helper em muitos pontos do códigos, vai ter que sempre ficar adicionando a linha do `import`. Para melhorar ainda mais, podemos configurar o Play para já adicionar alguns imports automáticos nas nossas views. Basta que o arquivo `build.sbt` seja alterado.

```
1      //codigo anterior
2
3      templatesImport += "infra.FormatadorDeDatas._"
```

Este é um recurso importante pois dessa maneira você não precisa, por exemplo, manter todos os seus `models` no pacote **models**. Crie a estrutura que você quiser e adicione os imports aqui.

Conclusão

Nesse capítulo vimos o poder do mecanismo de views oferecidos pelo Play. Tratando elas como simples classes, fica muito fácil realizar composições e reaproveitá-las durante o desenvolvimento do seu sistema. Manter a organização das views é muito importante para a manutenibilidade da sua aplicação.

Internacionalizando a aplicação

Até este momento, todos os textos da aplicação estão escritos diretamente nas páginas. Por exemplo, vamos analisar a view `menu.scala.html`.

```
1      <li class=""><a href="" class=""><i class="">Sobre</i></a></li>
2      <li class=""><a href="" class=""><i class="">Ical</i></a></li>
3      <li class=""><a href="" class=""><i class="">RSS</i></a></li>
4      <li class=""><a href="" class=""><i class="">Contato</i></a></li>
```

Se quiséssemos ter uma versão do Agendatech para inglês ou qualquer outra língua, uma nova página deveria ser criada. Claramente esse modelo não é o ideal, duplicar toda a página só por conta de um idioma diferente torna a manutenção do sistema quase que impossível. Por sinal, essa técnica da aplicação poder ser acessada por pessoas de diferentes nacionalidades, é conhecida como **Internacionalização**.

Externalizando strings

Como já era esperado, mais uma vez, o Play já vem com suporte nativo a internacionalização. Na verdade, até já usamos um pouco quando fizemos as configurações das mensagens de validação. Caso você não se lembre, deixamos essas mensagens em um arquivo chamado **messages**.

```
1      error.frownNow = A data não foi preenchida corretamente
2      constraint.email = O email deve ser válido, na moral
3      constraint.required = Preenchimento obrigatório
4      org.hibernate.validator.constraints.NotBlank.message = Tem que preencher
5      error.required = O campo não foi preenchido
```

Esse vai ser justamente o arquivo para externalizar nossas strings. Vamos usar o menu como exemplo.

```
1      sobre = Sobre
2      ical = Ical
3      rss = RSS
4      contato = Contato
```

Agora que as strings já foram configuradas, podemos recuperar estas informações direto da página.

```

1      <li class=""><a href="" class=""><i class="">@Messages("sobre")</i></a></li> \
2
3      <li class=""><a href="" class=""><i class="">@Messages("ical")</i></a></li> \
4
5      <li class=""><a href="" class=""><i class="">@Messages("rss")</i></a></li> \
6
7      <li class=""><a href="" class=""><i class="">@Messages("contato")</i></a></li>

```

Muito simples! A classe Messages é escrita em Scala, mas perceba que não causa nenhum impacto negativo para o programador que está escrevendo a view. Pense que é uma sintaxe diferente assim como a ::Expression Language:: no JSP. Um detalhe interessante é que caso a chave passada não seja encontrada, o Play vai utilizar a própria chave como texto a ser exibido.

O próximo passo é permitir que usuários que falam inglês possam entender os textos da nossa aplicação. E aí é que vem a diferença, ao invés de alterar a página é necessário apenas criar um novo arquivo. Seguindo a convenção do Play, o nome do arquivo deve ser **messages-locale do usuário**. Por exemplo, para usuários que falam a língua inglesa nos Estados Unidos, o nome do arquivo seria **messages.en-US**. Abaixo segue um exemplo:

```

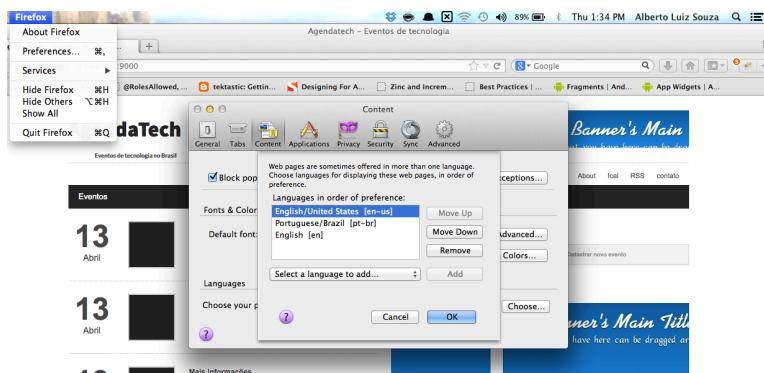
1      sobre = About
2      ical = Ical
3      rss = RSS
4      contato = Contact

```

Uma pergunta que pode estar passando pela sua cabeça é a de como o Play vai saber qual língua usar. Lembre sempre que a maneira mais comum de um cliente, no caso o navegador, passar informações relativas a configurações desejadas no servidor, é usando os headers do protocolo HTTP. Assim como foi feito para negociar o formato da resposta no capítulo “Servindo formatos diferentes”. Nesse exato momento, quando você faz uma requisição do seu navegador para a aplicação, ele está passando um header indicando qual a língua que ele prefere a resposta.

```
Accept-Language: en-us,pt-br;q=0.7,en;q=0.3
```

Como você pode ver o nome do header é **Accept-Language**. E perceba que várias opções de idioma são passadas, tudo vai depender de como seu navegador está configurado. Por exemplo no Firefox, para alterar essa configuração, basta que você acesse o menu **Preferences** e depois escolha a aba **Content**.



Você pode inclusive adicionar mais idiomas caso a sua aplicação realmente tenha que suportar diversas línguas. Quando a preferência de idioma é trocada, os requests automaticamente já são feitos com um novo valor no **Accept-Language**. O Play vai analisar as línguas prediletas e tentar achar a primeira que pode ser atendida pela sua aplicação.

Um ponto que você deve ficar atento é o número de strings externalizadas. Caso seja muito alto você vai precisar de uma estratégia de organização. A maneira mais comum é criar uma hierarquia, separando as strings por categorias. Um exemplo seria:

```
1 menu.sobre = Sobre
2 menu.ical = Ical
3 menu.rss = RSS
4 menu.contato = Contato
```

Dessa forma a busca por strings específicas dentro do arquivo fica facilitada. O nível de hierarquia vai depender da complexidade das suas páginas.

Nesse exato momento temos dois arquivos de mensagens, o **messages** e o **messages.en-US** e você deve estar se perguntando onde está o arquivo específico para português do Brasil. Perceba que antes, mesmo sem o arquivo da língua inglesa, as mensagens de validação estavam sendo respeitadas. É justamente para isso que serve o arquivo **messages**, o Play utiliza ele sempre que a língua indicada pelo navegador não for encontrada em nenhum outro arquivo. Este arquivo é chamado de **fallback**.

Um último detalhe, mas muito importante. Você precisa configurar o Play para aceitar os idiomas que você precisa. A fim de fazer isso, só precisamos mudar o arquivo **** application.conf ****

“ ‘ application.langs = “pt-BR,en-US,en”

“ ‘ Você especifica, respeitando a ordem de importância, quais idiomas a sua aplicação precisa lidar.

Adicionando parâmetros nas mensagens

Uma outra situação muito comum no momento de exibir as mensagens internacionalizadas é a necessidade que a mensagem seja construída dinamicamente. Por exemplo, na página de administração queremos dar boas vindas ao usuário que acabou de logar no sistema.

```
1  @(naoAprovados:List[Evento], aprovados:List[Evento])
2  <html>
3      <body>
4          <div>
5              @Messages("admin.bem_vindo")
6          </div>
7
8      ... resto do arquivo
```

E a mensagem no arquivo seria algo assim:

```
1  admin.bem_vindo = Bem vindo ??info_do_usuario
```

O Play usa por debaixo do pano as classes padrões do Java para carregamento de arquivos de mensagens, também chamado de bundles. Para especificar que a mensagem espera um parâmetro, devemos deixar especificado o índice do mesmo na mensagem.

```
1  admin.bem_vindo = Bem vindo {0}
```

Agora basta que seja passado o parâmetro necessário para a mensagem.

```
1  @Messages("admin.bem_vindo", session().get("email"))
```

Importante lembrar que caso o parâmetro não seja passado, o Play vai exibir **{0}**. Aqui foi utilizado a informação que está na sessão do Play, mas lembre que nela só pode ser armazenada objetos do tipo *String*. Caso você precise de um objeto *Usuario* completo, pode usar a API de Cache, que inclusive já foi vista no livro. O único cuidado é que você terá que lidar com as chaves do Cache para diferenciar os usuários logados na sua aplicação. Use o email como chave e tudo ficará certo.

Mudando o idioma programaticamente

Respeitar o idioma sugerido pelo navegador é uma ótima forma de tentarmos acertar a língua preferida pelo usuário. O problema dessa abordagem é se o idioma configurado não for o realmente preferido dele. Nesse momento, caso você esteja achando que todo mundo pode trocar isso facilmente, pense em todas as pessoas que você já conheceu que não tinham muito conhecimento de computação. Minha mãe por exemplo, apesar de usar diariamente um navegador, não saberia como fazer essa troca.

Uma estratégia adotada por alguns sites é a de oferecer links para que o próprio usuário possa definir o idioma predileto.

Latin America

Argentina / Spanish
Bolivia / Spanish
Brazil / Portuguese
Chile / Spanish
Colombia / Spanish
Costa Rica / English
Costa Rica / Spanish

Ecuador / English
Ecuador / Spanish
El Salvador / English
El Salvador / Spanish
Guatemala / English
Guatemala / Spanish
Honduras / English
Honduras / Spanish

Jamaica / English
Jamaica / Spanish
Nicaragua / English
Nicaragua / Spanish
Peru / Spanish
Panama / English
Panama / Spanish

Puerto Rico / English
Puerto Rico / Spanish
República Dominicana / English
República Dominicana / Spanish
Trinidad & Tobago / English
Trinidad & Tobago / Spanish
Venezuela / Spanish

Vamos então adotar a mesma abordagem para o Agendatech. Inicialmente vamos adicionar os links para que o usuário possa escolher. Faremos isso no arquivo **menu.scala.html**

```
1 <li class=""><a href="@routes.CookieLang.mudaIdioma?lang=pt-BR" class="">
2   <i class="">@Messages("idioma.pt_br")</i></a></li>
3 <li class=""><a href="@routes.CookieLang.mudaIdioma?lang=en-US" class="">
4   <i class="">@Messages("idioma.en_us")</i></a></li>
```

Perceba que já foi até escrito o controller que vai lidar com essa lógica. O seu console deve estar exibindo inclusive um erro de compilação informando que essa rota não existe.

Para que você não fique ansioso, vamos logo implementar o controller que executa a troca do idioma.

```
1 public class CookieLang extends Controller {
2
3   public static Result mudaIdioma(){
4     String lang = request().getQueryString("lang");
5     response().setCookie(Play.langCookieName(), lang);
6     return redirect(routes.EventosController.lista());
7   }
8 }
9 }
```

O Play, na versão Java, não possui uma maneira elegante de trocarmos o idioma. Mas isso não nos impede de irmos um pouco mais para o baixo nível. Repare que setamos o cookie que define a língua diretamente no **response**. O método `langCookieName` retorna o nome usado pelo Play para escrever o cookie.

Para finalizar, vamos adicionar essa action no **routes**.

```
1 GET          /idioma          controllers.CookieLang.mudaIdioma()
```

Um lembrete importante é que uma vez que o cookie da língua foi definido, o Play sempre vai dar preferência a ele em relação ao idioma enviado pelo navegador. Para voltar ao funcionamento original, o usuário terá que limpar os cookies do navegador ou sua aplicação terá que retirar essa entrada dos cookies. Caso opte pela segunda alternativa, você pode usar o código abaixo.

```
1 public static Result limpaIdioma(){
2     response().discardCookie(Play.langCookieName());
3     return redirect(routes.EventosController.lista());
4 }
```

Atenção com o Cache

No capítulo “Servindo formatos distintos” passamos um problema com o cache por conta que o header **Accept** não estava sendo respeitado. A solução encontrada foi a de controlar o cache manualmente. Agora o problema volta a tona. Com a nossa listagem sendo cacheada, caso o usuário troque o idioma, o mesmo não sera refletido na página.

A solução mais completa seria ter uma maneira de configurar os headers que devem ser levados em consideração no momento de fazer o cache. Mas como a implementação do Play não aceita isso, vamos buscar uma alternativa mais simples, pelo menos neste instante. Partes do sistema, onde as informações presentes nos headers tem influência sobre a resposta, não serão cacheadas. Então, vá no controller de eventos e retire o cache implementado. Uma outra alternativa seria escrever um plugin de Cache que levasse em consideração este tipo de situação.

Conclusão

Neste capítulo foi coberto a parte de internacionalização do Play. Algo simples, mas muito comum entre todas as aplicações. Mais uma vez tudo estava integrado ao framework, deixando muito fácil toda a configuração. Lembre sempre de tomar cuidado com as mensagens que dependem de parâmetros, a falta deles vai fazer a mensagem aparecer estranha na página.

Tome ainda mais cuidado com as actions que foram cacheadas, caso você esteja trocando o idioma e a troca não esteja sendo refletida na aplicação, tente dar uma olhada nesse aspecto!

Fazendo o deploy da aplicação

Nossa aplicação já tem a funcionalidade principal praticamente implementada. Eventos são cadastrados, moderados e listados na home. Com a parte de autenticação já feita, chegou a hora de instalarmos o projeto em algum servidor, processo também conhecido como deploy.

Executando em modo de produção

Até agora estamos acostumados a *rodar* o comando `play ~run` para executar a aplicação. Como você mesmo deve ter notado, toda alteração que era feita no código, tanto em classes quanto em arquivos de configurações, refletia automaticamente no sistema, o que é ótimo para rodar em modo de desenvolvimento. O problema disso para produção, é que a cada request, o Play realiza uma checagem completa para verificar se não tem nada de novo na aplicação, tornando a resposta do mesmo mais lenta do que deveria. Sem contar que o momento de deploy é muito importante no ciclo de vida de desenvolvimento, você não quer que sua aplicação seja atualizada sem a sua permissão.

Pensando nisso o framework já disponibiliza outra task específica para quem precisa rodar ele em modo de produção. Basta trocar o **run** pelo **start**. Quando o comando `play start` é executado, ele automaticamente corta todas as checagens que comentamos e sua aplicação realmente está rodando em modo de produção. O detalhe a ser notado nessa abordagem, é que você deve ter toda infraestrutura do Play no seu servidor de produção. Você vai precisar ter o sdk do Play instalado, além do código da sua aplicação. A maneira mais comum de vai fazer isso é instalando o seu sistema de versionamento de código no servidor. Por exemplo, caso sua empresa use o **git**, após o download do sdk, seu fluxo de trabalho para realizar um deploy seria basicamente o que segue:

- `git pull origin master`(baixaria a última versão do código)
- `cd pasta_do_seu_projeto`
- `play start`

Basta colocar isso num script e você consegue deployar o seu projeto em modo de produção. Perceba que não é necessário instalar um servidor separado, copiar seu projeto para uma pasta específica, nem nada disso. Rápido e indolor.

Caso você não tenha permissão para instalar o git e o sdk do Play no seu servidor, o que é até bastante comum, o framework já oferece um caminho alternativo. É possível gerar um **zip** com tudo necessário para rodar sua aplicação em modo **standalone**, basta que você rode o comando `play dist`. Esta task vai gerar um zip na pasta **target/universal** com o nome baseado nesse template: **nome_do_projeto-versao_no_build_sbt**. No caso do nosso projeto, o nome gerado foi *agendatech-1.0-SNAPSHOT.zip*.

Esse zip possui todas as dependências necessárias para rodar o seu projeto além de um script já pronto para subir a aplicação em modo de produção. Depois de extraído, basta que você execute o script da seguinte maneira: `agendatech-1.0-SNAPSHOT/bin/agendatech`. Perceba que o nome do script é o mesmo nome da sua aplicação. Esta maneira é ainda mais fácil do que a primeira, dado que você só precisa ter o Java instalado no seu servidor!

Configurando o servidor de produção

Nesse momento, não seria anormal se você estivesse com algumas perguntas dentro da sua cabeça. Vou tentar listar algumas aqui.

- Mas eu não quero rodar na porta **9000**, como faço?
- Quer dizer que vou usar o mesmo banco de desenvolvimento em produção?
- E as configurações, vou querer usar um email de verdade não o mock configurado.
- Caso eu troque de banco, o primeiro usuário vai ter que rodar as evoluções para mim?
- Será que consigo rodar o meu servidor com uma configuração diferente de memória?

Essas são preocupações mais que do que justas. Como é de praxe, você não precisa pensar muito para implementá-las. Todas essas configurações podem ser passadas como argumento para seu script de inicialização. Por exemplo, caso queira mudar a porta, o parâmetro que deve ser utilizado é o `-Dhttp.port=1234`.

```
1 agendatech-1.0-SNAPSHOT/bin/agendatech -Dhttp.port=80
```

Um ponto um pouco mais interessante é a troca do banco. Você já tem um arquivo base com diversas configurações, o necessário agora é apenas trocar algumas destas configurações. O primeiro passo para realizar isso é a criação de mais um arquivo, vamos chamá-lo de **production.conf**. A principal alteração que queremos fazer é a de utilizar outro banco para o sistema em produção.

```
1 db.default.url="jdbc:mysql://localhost/agendatech_play_producao"
2 logger.application=INFO
```

Além disso, em ambiente de produção é interessante diminuir o nível de log para que não seja criado arquivos muito grandes. Em sistemas pequenos isso não faz muita diferença, mas caso a aplicação que esteja na sua mente seja maior, grandes arquivos de log costumam criar muitos problemas.

Um problema que temos aqui é que ainda precisamos do resto das configurações deixadas no **application.conf**. Para não ter que copiar tudo de novo e ficar com código duplicado, o Play oferece uma maneira de você incluir as configurações de um arquivo em outro.


```
1      include "application.conf"
2
3      db.default.url="jdbc:mysql://localhost/agendatech_play_producao"
4      logger.application=INFO
```

Dessa maneira você pode inclusive modularizar suas configurações. Arquivo para configuração de autenticação, banco de dados, email, etc. Apenas lembrando que caso seja definida uma chave que já existe no arquivo incluído, a mesma é sobrescrita pela nova definição. Agora você pode inclusive colocar as configurações de email verdadeiras! Com o arquivo de produção criado, é necessário informar ao Play que queremos usá-lo ao invés do **application**. Basta que seja passado mais um parâmetro na inicialização.

```
1      agendatech-1.0-SNAPSHOT/bin/agendatech -Dhttp.port=80
2                                          -Dconfig.resource=production.conf
```

Também é possível passar um caminho que esteja fora do seu projeto. Útil para as situações onde a empresa tem uma organização particular para essas questões de configurações. O parâmetro seria **-Dconfig.file=caminhodoarquivo**.

Agora que o banco foi alterado, na primeira vez que a aplicação for executada, o Play vai tentar aplicar as evoluções para criar as tabelas. O problema é que seria muito estranho se um usuário da sua aplicação recebesse aquela tela onde o Play indica todas as evoluções que devem ser aplicadas! Ele conheceria todas tabelas do sistema, uma bela falha de segurança. Para ficar prevenido, vamos falar para o framework aplicar todas as evoluções no momento da inicialização da aplicação.

```
1      agendatech-1.0-SNAPSHOT/bin/agendatech -Dhttp.port=80
2                                          -Dconfig.resource=production.conf
3                                          -DapplyEvolutions.default=true
```

Você ainda pode trocar qualquer chave de que esteja presente no arquivo de configuração. Por exemplo, para passar a senha do banco pela linha de comando ao invés de deixar escrito no arquivo, poderíamos fazer o seguinte:

```
1      agendatech-1.0-SNAPSHOT/bin/agendatech -Dhttp.port=80
2                                          -Dconfig.resource=production.conf
3                                          -DapplyEvolutions.default=true
4                                          -Ddb.default.password=seupassword
```

Um último ponto que você pode precisar, é passar argumentos específicos da JVM para sua aplicação. Para fazer isso, o Play define uma convenção que indica que aquele argumento é para a JVM.

```
1  agendatech-1.0-SNAPSHOT/bin/agendatech -Dhttp.port=80
2      -Dconfig.resource=production.conf
3      -DapplyEvolutions.default=true
4      -Ddb.default.password=seupassword
5      -J-Xms512M -J-Xmx512m
```

Qualquer outra configuração necessária pode ser passada na hora da inicialização. Uma outra muito comum é a de indicar um arquivo de log diferente do padrão.

Escalando ainda mais

O Play sozinho já escala mais do que a grande maioria dos frameworks que temos no mercado. Inclusive um estudo feito pela empresa ZeroTurnaround, criadora do **JRebel**, mostrou que o Play é o framework com melhor performance e escalabilidade que temos hoje no mundo Java(<http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/>).

Mesmo com todo esse poder por conta da sua arquitetura totalmente assíncrona, pode ser que sua aplicação, devido a um aumento muito grande no número de requests, precise de mais poder de processamento. Casos comuns são as promoções feitas pela empresas aéreas, e-commerces, etc. Nessas situações a solução mais comum, e barata, é a de adicionar novos servidores para atender as requisições. Por conta da virtualização, criar novas máquinas tornou-se um trabalho simples. Empresas como Amazon ou DigitalOcean oferecem painéis online para controle dessas máquinas. Esta estratégia é conhecida como escalabilidade horizontal. Uma outra, também utilizada, é a de comprar uma máquina com uma configuração muito acima da média que suporte o aumento do número de requests, também conhecida como escalabilidade vertical.

Subindo vários servidores

Já que a técnica adotada é a da escalabilidade horizontal, a primeira tarefa a ser feita é a de subir algumas instâncias da sua aplicação. Para realizar o teste da sua própria casa, basta que você abra duas abas do seu terminal.



Agora é necessário que a sua aplicação seja iniciada nas duas. Lembre de trocar a porta do servidor.

```
1  agendatech-1.0-SNAPSHOT/bin/agendatech -Dhttp.port=8080
2      -Dconfig.resource=production.conf
3      -DapplyEvolutions.default=true
4
5
6  agendatech-1.0-SNAPSHOT/bin/agendatech -Dhttp.port=8081
7      -Dconfig.resource=production.conf
8      -DapplyEvolutions.default=true
```

Muito simples! Pensando um pouco mais sobre essa configuração, caímos em um problema. Como o usuário vai saber qual servidor acessar? Caso ele tenha que especificar a porta no endereço a ser digitado, subir duas instâncias não vai servir para nada, dado que o mesmo usuário vai ficar acessando sempre o mesmo servidor.

Load Balancer

Por isso que é necessário o uso de um terceiro servidor que conheça os endereços de todas as instâncias da sua aplicação. Esse servidor geralmente é chamado de **Load Balancer**. Existem algumas implementações deles, e as mais conhecidas são:

- Apache httpd com o seu plugin **mod_proxy**
- Nginx
- HAProxy

Para o exemplo deste livro será usado o HAProxy, por ser o único focado realmente em fazer o balanceamento de carga. Independente da escolha, você sempre vai ter que indicar quais são os endereços dos servidores responsáveis por atenderem as requisições. Por exemplo, no HAProxy temos que criar um arquivo com a extensão **.conf**.

```
1  global
2      daemon
3      maxconn 256
4
5  defaults
6      mode http
7      timeout connect 5000ms
8      timeout client 50000ms
9      timeout server 50000ms
10
11      option forwardfor
12
```

```
13             option http-server-close
14
15     frontend http-in
16         bind *:8080
17         default_backend servers
18
19     backend servers
20         balance roundrobin
21
22             option httpclose
23             option redispatch
24
25     server play1 127.0.0.1:8081 maxconn 32
26         server play2 127.0.0.1:8082 maxconn 32
```

O conteúdo da configuração não é o mais importante. Vamos analisar apenas alguns pontos.

- **timeout**: tempos de espera para as etapas que o HAProxy tem que seguir. Por exemplo, conseguir conectar num dos servidores configurados.
- **option forwardfor**: faz com que o HAProxy passe o IP original da requisição através do header “X-Forwarded-For”.
- **bind**: indica que o HAProxy vai escutar requisições na porta especificada. Geralmente a porta vai ser a **80**
- **server**: endereço do servidor que vai ser utilizada pelo HAProxy para repassar as requisições.

Agora que os dois servidores com o Agendatech já estão rodando e já temos a configuração para o Load Balancer, basta que o HAProxy seja iniciado. O script de inicialização dele só funciona para Linux e Mac, então caso você esteja no windows terá que baixar o Cygwin e realizar o download da instalação. Para os outros usuários, basta que seja usado o **apt-get** para o Linux ou o **brew** para Mac. Com o HAProxy instalado, podemos rodar o comando de inicialização.

```
1 haproxy -f haproxy.conf
```

Caso você rode o comando haproxy de outra pasta, basta passar o caminho do arquivo de configuração. Pronto, agora você pode navegar a vontade pelo Agendatech sem saber qual instância está servindo a sua aplicação. Caso a aplicação precise de mais instâncias, é só subir mais um servidor e adicionar a entrada no arquivo de configuração. Um outro ponto interessante também é que se algum servidor venha a ter algum problema, a aplicação continuará rodando. Isso é o que chamamos de disponibilidade.

Stateless ajudando na escalabilidade

Um detalhe interessante é que nossa aplicação guarda os dados do usuário logado na sessão. Utilizando um framework tradicional do mundo Java, esses dados ficariam num **HttpSession** que está vinculado ao servidor que atendeu a requisição de login. O problema dessa abordagem é que, depois do usuário ter logado, ele só pode ser direcionado para o mesmo servidor, pois os dados dele estão lá! Para contornar isso, os Load Balancers aceitam uma configuração onde pode ser feito um link entre o id da sessão do usuário e um servidor. Dessa forma, toda vez que este usuário tentar acessar a aplicação, vai ser direcionado para o mesmo servidor. Esse conceito é chamado de **Sticky Session**.

No caso do Play esse problema inexistente. Como não existe o conceito de sessão propriamente dito, já que todos os dados que guardados lá são colocados no cookie do cliente, você não precisa ter nenhum tipo de preocupação em relação a qual servidor vai atender a requisição. Os dados necessários para atender o request estão sempre vindo do cliente! Ser **stateless** faz com que sua aplicação realmente consiga tirar o máximo de proveito de todos os servidores que foram adicionados para atender os clientes.

Https

Https é a maneira padrão de criptografar as requisições entre o cliente e uma aplicação que utiliza o HTTP como protocolo de transporte. O ponto aqui não é entrar em detalhes sobre a especificação e sim como fazer que sua aplicação possa tirar proveito dela.

Para sua aplicação realmente ter uma conexão segura, sua empresa tem que adquirir um certificado de uma empresa que seja amplamente aceita pelos navegadores mais utilizados no mundo. Falo dos navegadores pois eles formam a maior parte das aplicações clientes dos seus sistemas web. Uma apresentação bem interessante sobre Https e os certificados pode ser encontrada nesse link(<http://www.youtube.com/watch?v=Z7Wl2FW2TcA>).

Caso a sua empresa tenha feito todo o processo necessário, você vai precisar informar ao Play onde está o arquivo gerado que contém o seu certificado e sua assinatura, além da senha utilizada para gerá-lo.

```
1  agendatech-1.0-SNAPSHOT/bin/agendatech -Dhttp.port=8080
2      -Dconfig.resource=production.conf
3      -DapplyEvolutions.default=true
4      *-Dhttps.port=9001*
5      *-Dhttps.keyStore=mykeystore.jks*
6      *-Dhttp.keyStorePassword=myspassword*
7
```

Perceba que é necessário especificar qual é a porta que sua aplicação vai atender as requisições utilizando **HTTPS**. Um detalhe interessante é que muitas vezes você quer adicionar o **HTTPS** mesmo

sem ter um certificado reconhecido. Caso comum para intranets por exemplo. Nesse cenário o Play também te ajuda. Ao invés de você ter que fazer isso na mão, utilizando o utilitário da *JDK* chamado **keytool**, basta que você passe a porta HTTPS e não passe mais nenhuma configuração.

```
1 agendatech-1.0-SNAPSHOT/bin/agendatech -Dhttp.port=8080
2 -Dconfig.resource=production.conf
3 -DapplyEvolutions.default=true
4 *-Dhttps.port=9001*
```

O Play automaticamente vai gerar um certificado assinado por ele e utilizá-lo para lidar com a criptografia da requisição. O arquivo vai ser gerado na pasta chamada **conf** e vai ser criado com o nome **generated.keystore**.

Deploy no cloud

Hoje em dia, além de clouds que oferecem infraestrutura como a Amazon e a DigitalOcean, também chamado de IAAS (Infrastructure as a service), existem os clouds que já oferecem toda uma plataforma pronta, como o Heroku, Cloudbees, OpenShift, etc, conhecidos como PAAS (Platform as a service). A principal diferença é que o segundo grupo já oferece tudo pronto para você fazer o deploy da sua aplicação. Servidor instalado, runtime da linguagem instalada, banco de dados e tudo mais.

E como o Play já é um framework bem conhecido no mercado, vários desses PAAS já oferecem integração pronta. Vamos olhar um exemplo usando o Heroku.

Deploy no Heroku

Inicialmente é necessário instalar o programa de linha de comando do Heroku. Um tutorial pode ser encontrado neste endereço <https://toolbelt.heroku.com/>. Uma vez que essa etapa está feita o processo é muito simples. O primeiro passo é criar uma nova aplicação que rode em Java no Heroku. Para isso, dentro da pasta do nosso projeto devemos rodar o seguinte comando:

```
1 heroku create agendatech-livro-play
```

O heroku já possui uma integração fina com o sistema de versionamento de arquivos **Git**, o qual este autor recomenda que seja usado pelo leitor. Para a instalação do Git siga este tutorial fornecido pelo Github (<https://help.github.com/articles/set-up-git>). Com o Git instalado, basta que você adicione o controle de versionamento a seu projeto. Dentro da pasta do projeto, execute os seguintes passos:

- `git init`
- `git add -A`
- `git commit -m "commit inicial do projeto"`

Agora basta enviar o código da sua aplicação para o Heroku!

```
1 git push heroku master
```

Uma vez que sua aplicação tiver sido colocada no ar, basta que você acesse através do nome de criação da mesma (agendatech-livro-play.herokuapp.com). Um detalhe importante é checar a configuração de banco necessária. Veja na documentação do Heroku.

Esse é um ponto a favor dos clouds. A facilidade ficou tão grande que fica difícil decidir se é melhor criar uma infraestrutura própria ou tirar proveito dessas funcionalidades.

Conclusão

Este capítulo inteiro foi dedicado ao momento de deploy da sua aplicação. Geralmente esse que deveria ser um momento de felicidade para os programadores, afinal de contas o trabalho finalmente será publicado, vira um momento de tensão por conta dos passos necessários para a instalação da aplicação.

O Play tentou minimizar bastante este trabalho, pelo menos no que diz respeito à aplicação em si. Tudo necessário para produção pode ser passado via argumento, o que facilita bastante. Uma vez que a linha de comando é colocada num arquivo de execução, você não precisa mais alterar. Outros pontos interessantes em relação a automatização de deploy são:

- Backup do banco em intervalos regulares
- Backup da aplicação que está em produção no momento
- Configuração da máquina que roda a aplicação para fácil replicação.

Até uma próxima

Com o que você já tem agora, é capaz de desenvolver qualquer aplicação! Não espere muito, tente usar o framework no seu próximo projeto, afinal de contas a prática leva a perfeição.

Obrigado por ter dedicado seu tempo lendo este livro. Espero que a leitura tenha sido tão proveitosa quanto foi a escrita para mim. Qualquer dúvida que você tenha fique a vontade para postar no forum criado pela Leanpub, eu estarei sempre atento. Você também pode me encontrar no twitter através do usuário [@alberto_souza](https://twitter.com/alberto_souza). Estou sempre colocando links interessantes sobre o Play.

Os próximos capítulos serão de partes mais específicas do framework como os plugins e as evolutions.

Reaproveitando funcionalidades através de plugins

Uma das funcionalidades implementadas no Agendatech é a de mandar email para cada novo evento que é cadastrado no sistema. Como enviar emails é uma tarefa muito comum, ao invés de implementar esse código na mão, foi feito uso de um plugin específico para mandar emails. O uso de plugins constitui uma parte muito importante dos seus projetos, pois inúmeras vezes a funcionalidade que você vai implementar já está pronta!

Por exemplo, o código de upload que escrevemos para cadastrar uma foto associada ao evento foi bastante penoso. Vamos dar uma olhada para refrescar a memória.

```
1     private static File gravaDestaque() throws IOException {
2         MultipartFormData body = request().body().asMultipartFormData();
3         FilePart filePart = body.getFile("destaque");
4         if (filePart == null)
5             return null;
6
7         File destaque = filePart.getFile();
8         File destino = arquivoDeDestino(filePart);
9         FileUtils.moveFile(destaque, destino);
10        return destino;
11    }
12
13    private static File arquivoDeDestino(FilePart destaque) {
14        return new File("public/images/destaques", System.currentTimeMillis()
15            + "_" + destaque.getFilename());
16    }
```

Tivemos que lidar com a gravação do arquivo e com a definição do nome do mesmo. Além disso, um detalhe que não nos preocupamos, é com o fato de que provavelmente vamos querer tratar essa imagem que foi enviada pelo usuário. Por exemplo, deixá-la de um tamanho que seja adequado ao layout da nossa aplicação. Para não termos que fazer isso todas as vezes, vamos criar um plugin que isole esse código e nos permita reaproveitá-lo entre vários projetos.

Utilizando e implementando o plugin

Antes de entrarmos na implementação, vamos dar uma olhada no exemplo para o nosso plugin. É sempre importante lembrar que, na construção de uma API para ser usada por várias pessoas, a parte

mais importante são os métodos que estão expostos para serem usados. Como os mesmos vão ser implementados, não é tão importante para a gente. Vamos começar focando nisso!

O primeiro ponto que o usuário do nosso plugin vai precisar ter acesso é a classe que expõe para ele as funcionalidades da nossa extensão.

```
1  import play.Play;
2  ...
3
4  PaperClipPlugin plugin = Play.application().plugin(PaperClipPlugin.class);
```

O método `plugin` funciona como um *Factory Method*. Perceba é passado como parâmetro a classe que representa seu componente e o método retorna um objeto pronto para uso. Nada muito trabalho, como é de praxe. A classe *PaperClipPlugin* precisa apenas implementar a interface *play.api.Plugin*.

```
1  public class PaperClipPlugin implements Plugin{
2
3      private Application application;
4
5      public PaperClipPlugin(Application application) {
6          this.application = application;
7      }
8
9      @Override
10     public boolean enabled() {
11         return true;
12     }
13
14     @Override
15     public void onStart() {
16         Logger.info("Starting PaperClip plugin");
17     }
18
19     @Override
20     public void onStop() {
21         Logger.info("Stopping PaperClip plugin");
22     }
23 }
```

Somos obrigados a receber o parâmetro do tipo *Application* no construtor. Através desse parâmetro você pode saber, por exemplo, se está em ambiente de desenvolvimento ou produção.

Uma das funcionalidades do nosso plugin é a de permitir que uma foto seja recortada no centro. Segue um exemplo:

```
1 UploadedImage image =  
2     UploadedImageFormatter.toUploadedImage(request(), "destaque");  
3 UploadedImage croppedImage = plugin.centeredCrop(image, 200, 200);  
4 croppedImage.save("destaques");
```

O componente já fornece o método que faz o recorte. O detalhe a mais é que devemos transformar o arquivo que veio do upload em objeto do tipo *UploadedImage*. Aqui podemos até melhorar e transformar esse **Formatter** num conversor padrão do Play. Dessa maneira poderíamos ter um atributo desse tipo no nosso modelo. Perceba que toda aquele código confuso de upload foi embora, temos uma implementação bem mais simples e que pode ser reaproveitada em outros projetos.

Na próxima seção já vamos falar sobre isso, mas o ideal é que o código do plugin esteja em outro projeto, justamente para o mesmo poder ser reaproveitado entre várias aplicações. Dado esse contexto, precisamos notificar o Play que o plugin deve ser carregado. O arquivo responsável por fazer isto é o **play.plugins**. Inclusive o mesmo já foi usado quando o plugin de email foi configurado.

```
1 1500:com.typesafe.plugin.CommonsMailerPlugin  
2  
3 10500:com.github.asouza.play.paperclip.PaperClipPlugin
```

Apenas para lembrar, o número que vem a frente tem a ver com a ordem de carregamento dos plugins. Acima de **10000** significa que queremos que ele seja carregado depois da execução da classe **Global**, que está na raiz do projeto. Caso esteja curioso, acesse esse link(<http://bit.ly/1gVbHl7>) e veja alguns dos plugins que são carregados quando a aplicação é iniciada.

Pronto, agora já podemos usar tranquilamente o plugin de upload nos nossos projetos. O código completo está no github. Para verificar, acesse esse endereço(<https://github.com/asouza/play-paperclip>).

Detalhes na implementação de um plugin

Para a criação de um plugin, algumas precauções devem ser tomadas. Como citado, o ideal é que o código fique em outro projeto, para poder ser reutilizado. O problema aqui é que você vai precisar das dependências do Play. A maneira mais fácil de resolver esta situação é simplesmente criando um novo projeto usando o Play! Dessa forma tudo já vai estar configurado e você só precisa preocupar-se com a sua implementação.

Um outro detalhe que deve ser cuidado são os arquivos da pasta de configuração. Tudo que está lá dentro fica disponível no **classpath** da aplicação e caso você deixe, por exemplo, o arquivo **routes** lá dentro, pode ser que este seja usado ao invés do **routes** da aplicação que está utilizando o plugin. Geralmente na pasta **conf** do seu plugin devem ficar arquivos de configurações específicos, como os arquivos de internacionalização de chaves usadas pelo mesmo.

Publicando

Uma vez que o plugin está escrito, é importante que você consiga testá-lo na sua aplicação localmente e, posteriormente, libere ele para ser utilizado por outros projetos ao redor do mundo.

Seguindo a maneira comum, vamos adicionar a dependência no arquivo **build.sbt**

```
1      libraryDependencies += Seq(  
2          #outras libs,  
3          "com.github.asouza.play" %% "play-paperclip" % "1.0-SNAPSHOT"  
4      )
```

Analisando um pouco melhor, como o sbt vai resolver essa dependência se não publicamos ainda em nenhum lugar? Para resolver a primeira parte do problema vamos publicá-lo apenas localmente. No console da aplicação do seu plugin, basta que seja digitado `publishLocal`. Isso já vai registrar o plugin no repositório local da instalação do Play. Um detalhe a mais que você precisa adicionar no seu arquivo de build, é o nome da organização. Por exemplo, o de upload e qualquer outro que este autor venha a escrever, ficará sob a organização **com.github.asouza.play**.

```
1      #resto do arquivo  
2  
3      organization := "com.github.asouza.play"
```

Depois de testá-lo localmente, você provavelmente vai querer liberar seu plugin para o mundo. Para fazer isso temos apenas que fazer umas configurações a mais no arquivo de build.

```
1      #resto do arquivo  
2  
3      publishTo := Some("Sonatype repo" at  
4          "https://oss.sonatype.org/releases"),
```

A Sonatype é famosa por abrigar várias libs amplamente usadas nos nossos projetos. Além de indicar o endereço de publicação, precisamos ter as credenciais necessárias para publicar a biblioteca. Para a sua publicação ser bem sucedida, você precisará realizar as configurações necessárias na Sonatype. O guia completo pode ser encontrado em <http://bit.ly/1kxnTYx>.

Com tudo configurado, basta que você acesse o console do plugin e digite `publish`.

Conclusão

O plugin é uma forma poderosa de criarmos componentes reutilizáveis. Muito do próprio Play é implementado como plugin. Por exemplo:

- Associação das rotas com as actions
- Carregamento da classe *Global*
- Execução das evolutions.

Tire proveito dessa funcionalidade, tente deixar o máximo de código de infra reaproveitável em plugins. Dessa forma o seu projeto vai estar sempre muito mais focado nas regras de negócio.

Cuidando da evolução do banco

Por alguns momentos durante o desenvolvimento da nossa aplicação, fomos obrigados a criar tabelas e novas colunas no banco de dados. Para contemplar essa parte tão importante do processo de desenvolvimento, o Play fornece para a gente o mecanismo das **evolutions**, por sinal até já escrevemos algumas!

O leitor que conhece um pouco de Hibernate pode estar pensando qual vantagem as evolutions fornecem sobre o mecanismo de atualização de tabelas que já vem embutido dentro do próprio Hibernate. Para o leitor que não é familiarizado com Hibernate, o mesmo fornece uma configuração que o faz ir criando as tabelas e novas colunas no banco de dados, na medida que vamos alterando nossos modelos anotados com @Entity.

```
1 <property name="hibernate.hbm2ddl.auto" value="update"/>
```

Esse é um bom questionamento e merece alguns itens para reflexão, vamos imaginar algumas situações.

Na definição da classe Evento tivemos a preocupação de adicionar uma annotation no atributo descrição para informar que ele deveria representar uma coluna do tipo **text** no banco de dados.

```
1 @Entity
2 public class Evento {
3     ...
4     @Column(columnDefinition = "text")
5     private String descricao;
6 }
```

Caso não tivéssemos feito isso, agora seria a hora. O problema é que essa alteração não seria refletida no banco, já que ele não altera o que já foi criado. Provavelmente você teria que fazer um script sql para realizar essa alteração. E quando o outro desenvolvedor do time atualizasse o projeto, como que ele ia saber que tinha de alterar o banco de desenvolvimento dele?

Uma outra situação é a necessidade de deletar um atributo da sua classe. Como sua aplicação não precisa mais manter aquele estado, qual a necessidade de manter esse mesmo estado no banco de dados? Mais uma vez você cai no problema de ter que escrever um script sql para aplicar essa alteração no banco. E mais uma vez o outro desenvolvedor vai ter que descobrir que uma alteração deve ser aplicada na versão local do banco dele.

E mesmo que seu time consiga resolver todos esses pontos levantados, como essas alterações vão entrar em produção? Geralmente o banco de produção está em um ambiente controlado onde as alterações devem ser aplicadas manualmente e não diretamente através da aplicação.

Por esses motivos, deixar que o framework de persistência fique alterando o banco em função do modelo serve até razoavelmente bem para desenvolvimento de sistemas pequenos, mas logo fica muito confuso para sistemas um pouco maiores. E a perda de visibilidade das alterações é um dos maiores motivos.

Já existem algumas ferramentas no mercado que atacam justamente esse problema. E o Play, como framework full stack, já conta com a dele embutida.

Evolutions no Play

A nossa aplicação já vem evoluindo o banco de dados através de criação de arquivos sql.

```
1      # --- !Ups
2
3      alter table evento add data_de_inicio date;
4      alter table evento add data_de_fim date;
5
6      # --- !Downs
7
8      alter table evento drop column data_de_inicio;
9      alter table evento drop column data_de_fim;
```

Repare que o arquivo é dividido em duas partes: o comentário **!Ups** define o trecho que deve evoluir o banco, enquanto que o **!Downs** define o trecho que deve desfazer a evolução. Como tudo relativo a configuração, esses arquivos ficam dentro da pasta *conf* do projeto. Ainda dentro da pasta *conf*, foi criada a pasta *evolutions/default*, que de fato contém os scripts sql. O nome “default” vem do fato de termos configurado o servidor do *Ebean* com esse mesmo nome.

```
1 ebean.default="models.*"
```

Caso a aplicação trabalhasse com dois bancos de dados diferentes, você poderia ter outras pasta dentro de *evolutions*.

Para garantir a ordem de execução das evolutions, de uma maneira que seja fácil de entender pelo desenvolvedor, é utilizado um número no início de cada arquivo. Por exemplo para a evolution que adicionou as datas no evento, o nome escolhido foi *2.sql*. Idealmente, após o número, poderia inclusive ter sido usado um texto mais descritivo.

Qual evolution precisa ser executada?

Quando um outro desenvolvedor baixa as últimas atualizações do código e vai rodar a aplicação, o framework precisa decidir quais das evolutions ainda faltam ser aplicadas. Para realizar esse controle existe uma tabela criada exclusivamente para isso.

Field	Type	Null	Key	Default
id	int(11)	NO	PRI	NULL
hash	varchar(255)	NO		NULL
applied_at	timestamp	NO		CURRENT_TIMESTAMP
apply_script	text	YES		NULL
revert_script	text	YES		NULL
state	varchar(255)	YES		NULL
last_problem	text	YES		NULL

Como ela é de uso interno, o próprio Play realiza a criação da mesma de maneira automática. Para cada evolution aplicada, é criado um registro nessa tabela com os dados da mesma, dessa maneira o framework consegue saber quais já foram aplicadas e quais ainda faltam.

Problemas comuns

O modelo de nomenclatura das evolutions tem funcionado muito bem até agora. Colocar um número no início do arquivo, indica claramente a ordem que as evolutions devem ser executadas. Um possível problema é se você estiver trabalhando num time com mais desenvolvedores. Pense na seguinte situação: o sistema agora precisa ter duas novas funcionalidades. São elas: associação de um evento a um grupo responsável por sua organização e cadastramento de banners que podem ser exibidos no Agendatech.

Você fica responsável por implementar a primeira e outro desenvolvedor, vamos chamá-lo de Wolverine, fica responsável por fazer a parte dos banners. Wolverine, na máquina dele, cria uma nova evolution para ser aplicada no banco de dados, por exemplo *6.sql*.

```
1  # --- !Ups
2
3  create table banner (
4      id                integer auto_increment not null,
5      nome              varchar(255),
6      constraint pk_evento primary key (id))
7  ;
8
9  # --- !Downs
10
11  SET FOREIGN_KEY_CHECKS=0;
12
13  drop table banner;
14
15  SET FOREIGN_KEY_CHECKS=1;
```

Como é de praxe em sistemas com muitos desenvolvedores, a equipe usa algum controle de versão, GIT por exemplo. Após realizar sua implementação, Wolverine envia suas alterações para o servidor. Do seu lado, você também começou a implementar sua funcionalidade e também criou uma nova evolution, chamada *6.sql*. Antes de enviar suas alterações para o servidor, você é obrigado a fazer o download das últimas alterações, e aí que reside o problema. O GIT vai acusar um problema de conflito o qual o desenvolvedor é obrigado a resolver. Provavelmente, depois de resolver o conflito, sua evolution ficou desse jeito.

```
1  # --- !Ups
2
3  create table grupo (
4      id                integer auto_increment not null,
5      nome              varchar(255),
6      constraint pk_evento primary key (id))
7  ;
8
9  create table banner (
10     id                integer auto_increment not null,
11     nome              varchar(255),
12     constraint pk_evento primary key (id))
13 ;
14
15 # --- !Downs
16
17 SET FOREIGN_KEY_CHECKS=0;
18
```



```

19 drop table grupo;
20 drop table banner;
21
22 SET FOREIGN_KEY_CHECKS=1;

```

Quando você tentar acessar de novo a aplicação, o Play vai perceber que essa evolution já foi aplicada, mas que a mesma sofreu alguma alteração. Baseado nisso, ele vai mostrar a tela de erro sugerindo que você aplique de novo.



Basta clicar no botão de aplicação da evolution e seu problema vai estar resolvido. Uma maneira interessante de tentar evitar este tipo de situação é colocar mais alguma descrição no nome do arquivo ou até mesmo usar um **timestamp** no nome para diminuir as chances de conflito. Tome apenas cuidado com o número que vai ser gerado pois o Play, atualmente, suporta apenas o tipo **int** para os números das evolutions. Infelizmente, um timestamp geralmente é um **long**.

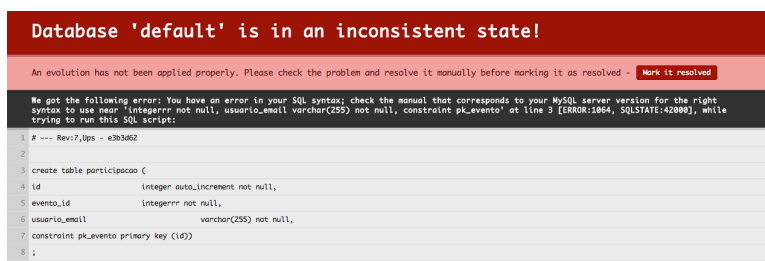
Uma outra situação muito comum também é a evolution conter algum sql inválido. Por exemplo:

```

1 # --- !Ups
2
3 create table participacao (
4   id integer auto_increment not null,
5   evento_id integerrr not null,
6   usuario_email varchar(255) not null,
7   constraint pk_evento primary key (id))
8 ;

```

Nesse caso, o tipo **integerrr** está escrito de maneira equivocada. Como é o default, quando a aplicação for acessada, o Play vai tentar aplicar a evolution.



E como era esperado, como o script está com um erro de sql, a evolution não foi executada com sucesso. Como a própria mensagem sugere, a evolution deve ser corrigida manualmente. Após isso, basta tentar acessar a aplicação que vai ser exibida a tela padrão de aplicação de evolutions.

Outras ferramentas

A implementação de evolutions default do Play já ajuda bastante durante o desenvolvimento. Apenas como fim de curiosidade, é interessante saber que existem outras ferramentas com o mesmo objetivo. Por exemplo, alguns usuários reclamam que dentro do Play não existe uma maneira de se aplicar essas evolutions por linha de comando, ou até mesmo de maneira programática. Outro ponto não implementado é falta de possibilidade de se fazer rollback de alguma evolution aplicada. Caso a sua aplicação precise de algumas dessas funcionalidades você pode tentar algumas das ferramentas listadas abaixo.

- Flyway(<http://flywaydb.org/>)
- Mybatis migrations(<http://mybatis.github.io/migrations/>)
- Liquibase(<http://www.liquibase.org/>)

Inclusive o Flyway já possui um plugin para integração com o SBT e, como o SBT é a ferramenta de build do Play, você pode usá-lo dentro do seu projeto. O mesmo pode ser encontrado nesse link(<http://flywaydb.org/blog/flyway-2.3.html>).

Conclusão

O objetivo deste capítulo foi fornecer mais alguns detalhes sobre as evolutions do Play. O ponto que merece mais atenção é a nomenclatura que vai ser utilizada nos arquivos. Lembre que em um ambiente com mais de um desenvolvedor, o que é comum, você precisa ter uma estratégia de nomes boa o suficiente para minimizar o número de conflitos. Usar o timestamp como início do nome tende a ser uma boa solução.

Hora de praticar

Muito obrigado por ter ficado comigo durante toda a jornada do livro! É muito importante que você tente praticar tudo que foi estudado no livro. Tem chance de escolher a tecnologia do novo projeto? Não fique acanhado, use o Play e foque nas suas regras de negócio entregando muito mais valor para o seu cliente. Sem contar que, mesmo esse ditado sendo velho, a prática leva a perfeição. Em cada novo projeto, você vai encontrar novos desafios que farão com que você entenda cada vez mais sobre o framework.

Mantenha contato

Eu estou completamente disponível para ajudá-lo onde for possível. Você pode me achar das seguintes formas:

- Por email, basta enviar uma mensagem para playnapratica@gmail.com
- Caso use o twitter, é só acessar https://twitter.com/alberto_souza
- Ainda tem a possibilidade de acompanhar meus projetos pelo github, basta acessar <https://github.com/asouza>.
- Acesse também meu blog e acompanhe diversos posts sobre o Play. O endereço é <http://alots.wordpress.com/>.

Por último, todo e qualquer feedback sobre o livro é bem vindo, a Leanpub inclusive fornece uma página para feedback. Você pode acessá-la através do endereço <https://leanpub.com/playframeworknapratica/feedback>. Caso queira mandar diretamente para meu e-mail, fique a vontade também. Escrever o livro de maneira independente foi um grande desafio, espero que ele tenha sido de alguma valia para você!

Consultoria e eventos

Precisando começar com o Play na sua empresa? Entre em contato! Terei o maior prazer em ajudá-lo.