



Rails para sua Diversão e Lucro

Copyright © 2006 Ronaldo Melo Ferraz

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Permissão é dada para copiar, distribuir e/ou modificar esse documento sob os termos da Licença de Documentação Livre GNU, Versão 1.2 ou qualquer outra versão posterior da mesma publicada pela Free Software Foundation; sem Seções Invariantes, sem Textos de Capa Frontal, e sem Textos de Quarta Capa. Uma cópia da licença está incluída na seção cujo título é "GNU Free Documentation License".

"Rails", "Ruby on Rails", e o logotipo do Rails são marcas registradas de David Heinemeier Hansson, com todos os direitos reservados. Os nomes e logotipo são usados aqui com permissão. Todas as demais marcas registradas são propriedade de seus respectivos donos, com todos os direitos reservados.

Conteúdo

Rails para sua Diversão e Lucro.....	1
Aqui há rubis.....	4
Pás e picaretas.....	5
Começando a aplicação.....	7
Primeiros passos.....	7
Configurando o banco de dados.....	10
Gerando migrações e modelos de dados.....	13
MVC.....	22
O primeiro controller.....	23
Layouts.....	27
Roteamento.....	32
Scaffolding.....	34
Validações.....	41
Um segundo controller.....	44
Extendendo um modelo de dados.....	62
Usando helpers.....	63
Relacionamentos.....	69
Belongs to.....	75
Has many.....	87
Has one.....	89
Has many, through.....	90
Avançando no Rails.....	91
Filtros.....	91
Uploads.....	107
Callbacks.....	118
Plugins.....	119
Has and belongs to many.....	123
Escopos.....	134
Filtros around.....	141
XML.....	148
Autenticação HTTP.....	150
Enviando e-mails.....	155
Depurando aplicações.....	160
Associações polimórficas.....	162
Módulos.....	165

Ajax.....	170
Agindo como uma lista.....	195
Transformando um modelo de dados.....	199
Continuando com Ajax.....	204
Degradação em Ajax.....	225
Avançando um pouco mais.....	227
Roteamento avançado.....	228
Caching.....	233
Tradução, Unicode e Globalização.....	251
Segurança.....	256
Unit Testing.....	260
Implantação.....	275
Ambientes de Desenvolvimento.....	278
Rake a seu serviço.....	279
Vivendo no limite.....	280
O que falta.....	281
Conclusão.....	281
Apêndice A: GNU Free Documentation License.....	282
0. PREAMBLE.....	282
1. APPLICABILITY AND DEFINITIONS.....	283
2. VERBATIM COPYING.....	284
3. COPYING IN QUANTITY.....	284
4. MODIFICATIONS.....	285
5. COMBINING DOCUMENTS.....	287
6. COLLECTIONS OF DOCUMENTS.....	287
7. AGGREGATION WITH INDEPENDENT WORKS.....	287
8. TRANSLATION.....	288
9. TERMINATION.....	288
10. FUTURE REVISIONS OF THIS LICENSE.....	288
How to use this License for your documents.....	288

Aqui há rubis

Há pouco mais de dois anos atrás eu descobria o Rails.

Procurando por um *wiki* para uso pessoal, eu topei com uma aplicação escrita em Ruby, uma linguagem de programação que—apesar da minha paixão pelo assunto—eu conhecia apenas por meio de menções esporádicas no *blog* do um amigo. A aplicação era o Instiki e eu a instalei minutos depois de terminar de ler suas características, que me atendiam perfeitamente.

Mais tarde, investigando o código do Instiki, eu descobri que ele fora desenvolvido em um novo *framework* para aplicações Web chamado Ruby on Rails, ou simplesmente Rails. A despeito de meu interesse antigo por todo e qualquer *framework* Web, foi somente após um mês que eu me decidi verificar o que era o Rails.

Dois anos depois, quase todo o meu desenvolvimento Web atual é feito em Rails e eu ainda uso o Instiki—ironicamente, a mesma versão que eu baixei há tanto tempo atrás.

Quando comecei a usar o Rails, uma das primeiras coisas que eu fiz foi elaborar um tutorial para ajudar as pessoas que estavam curiosas sobre o assunto a se familiarizarem com o ambiente. O tutorial era bem simples e atendia a um propósito básico: mostrar a alguém que já conhecia programação Web o que era esse novo *framework* do qual todo mundo estava falando e o que ele realmente podia fazer. Desde que liberei esse tutorial, ele continua sendo um dos arquivos mais baixados do meu servidor e, julgando pela quantidade de e-mails que eu ainda recebo sobre o mesmo, acredito que o objetivo proposto no mesmo foi cumprido plenamente.

O presente tutorial tem um novo objetivo: ir além do básico e mostrar mais do que o Rails é capaz de fazer, considerando principalmente o que foi introduzido a partir da versão 1.1 do mesmo. O tutorial anterior foi elaborado com base na versão 0.9.3 e desde então muita coisa mudou; várias novas facilidades foram introduzidas tornando o Rails ainda mais poderoso e flexível, levando o mesmo a um novo patamar de produtividade, que inclui a possibilidade de aproveitar plenamente os recursos da tão falada Web 2.0.

Um segundo objetivo é que esse tutorial também possa servir de base aos que quiserem utilizá-lo com referência em um curso de 8 a 16 horas, tratando de todos os tópicos necessários para um conhecimento—ainda que inicial—dos principais recursos de Rails e servindo de alavanca para um conhecimento mais avançado.

Com base nisso, o tutorial procura equilibrar uma introdução completa ao Rails com uma panorâmica de tópicos mais elaborados que incluem, entre outros, a construção de extensões para o *framework*, o uso de relacionamentos mais sofisticados e uma apresentação de assuntos relacionados ao uso de Ajax.

Como o tutorial possui esses dois objetivos distintos, muito do que é dado no começo do mesmo é uma elaboração bem mais completa do primeiro tutorial, atualizando o texto para a versão 1.1 do Rails. Sendo

assim, se você já tem o conhecimento básico necessário e deseja pular o já sabe, sinta-se à vontade para fazer isso. Entretanto, como muitas coisas mudaram entre as versões do Rails, alguma seções contém material novo e algumas considerações a mais que podem ser de alguma valia—um exemplo é a seção sobre relacionamentos entre modelos de dados, que fala também de alguns tipos mais avançados de relacionamentos introduzidos a partir da versão 1.1.

Dentro de ambos os objetivos, esse tutorial é voltado para desenvolvedores já familiarizados com algum ambiente de desenvolvimento Web, seja usando Linux ou Windows. Por razões de tempo, o tutorial assume que o desenvolvedor já tenha conhecimento de pelo menos uma linguagem de programação, de um banco de dados relacional qualquer e que seja capaz de estabelecer um ambiente para essa linguagem e banco de dados em sua plataforma de escolha, ou seja, que tenha o conhecimento técnico para acompanhar os passos dados no tutorial sem maiores problemas. O tutorial não assume, porém, que o desenvolvedor saiba fazer o mesmo para o Rails e explica isso nos detalhes necessários.

Um certo conhecimento do Ruby é requerido, embora a linguagem seja simples e poderosa o suficiente ser inteligível para qualquer desenvolvedor com uma experiência razoável em outra linguagem de alto nível. Apesar disso, esse tutorial pode não ser tão fácil para desenvolvedores que estejam dando os seus primeiros passos em programação e em especial programação para a Web—embora eu acredite que desenvolvedores iniciantes possam se beneficiar muito com o contato com uma linguagem mais poderosa e com um *framework* mais elaborado.

Uma segunda ressalva é que o tutorial foi escrito sob o Linux, utilizando-se das particularidades dessa plataforma. Onde houver necessidade, diferenças para o Windows serão notadas, mas as convenções seguirão o comum ao Linux. Dito isso, é perfeitamente possível aproveitar o tutorial em qualquer plataforma suportada pelo Rails sem mais do que a necessidade de mudar um ou dois detalhes na execução dos comandos e no código.

Depois de todas as considerações acima, meu conselho é somente um: divirta-se. O criador do Rails, David Heinemeier Hansson, já disse em várias ocasiões que um dos seus objetivos ao criar o Rails foi devolver aos programadores a diversão na hora de programar. Eu diria que ele conseguiu isso. Então, seja qual foi o seu propósito ao ler esse tutorial, aproveite bastante.

Pás e picaretas

Como dito acima, o presente tutorial assume que você tenha familiaridade com programação Web em geral, e com pelo menos um banco de dados relacional. Eu usei o MySQL na elaboração do tutorial por ele ser um banco de fácil instalação, mas você pode usar o que mais se adaptar ao seu ambiente, principalmente considerando que o Rails esconde a maior parte da complexidade nessa área e que ele suporta os principais bancos de dados existentes no mercado.

Para começar, vamos assumir que você tenha um banco de dados instalado e que esteja pronto para instalar o Rails e quaisquer outras bibliotecas necessárias. O primeiro passo então é instalar o Ruby, a linguagem no

qual o mesmo foi desenvolvido.

O Ruby é uma linguagem de alto nível, tão completa quanto uma linguagem poderia ser. Sua sintaxe é bem simples, e lembra Ada e Eiffel, com algumas pitadas de Perl em alguns pontos. Como o objetivo desse tutorial não é explicar ao Ruby, se você acha que precisa de uma introdução à linguagem, eu recomendo que você leia o tutorial escrito pelo Eustáquio “TaQ” Rangel, disponível no endereço seguinte:
<http://eustaquiorangel.com/files#ruby>.

O TaQ é conhecido de longa data, e seu tutorial é provavelmente a melhor introdução ao Ruby em português. Aproveitando para fazer uma média, ele também publicou um dos primeiros livros sobre a linguagem em nosso idioma. Você pode encontrar mais informações sobre o livro no seguinte endereço:
<http://www.braspert.com.br/index.php?Escolha=8&Livro=L00187>.

Voltando à instalação do Ruby, o processo é bem simples.

No Windows, a melhor maneira é usar o Ruby One-Click Installer, cuja versão atual é a 1.8.5. Esse instalador pode ser baixado de <http://rubyforge.org/projects/rubyinstaller/> e basta rodá-lo para ter um ambiente Ruby completo para o Windows, incluindo documentação e editores de texto.

Em distribuições do Linux como o Ubuntu ou Debian, com facilidades para a instalação de aplicativos, uma maneira rápida é usar as ferramentas da própria distribuição para baixar os pacotes e dependências necessárias. Um problema, entretanto, é que normalmente a versão do Ruby disponível nessas distribuições é a 1.8.2, que não será mais suportada pelo Rails em breve. O presente tutorial roda perfeitamente sob essa versão mas foi basicamente desenvolvido sob a versão 1.8.4.

Assim, se você está utilizando o Linux, eu recomendo que você compile a sua própria versão do Ruby, seja a 1.8.4 ou a 1.8.5. A versão 1.8.5 é a estável atualmente. Para compilá-la, utilize o procedimento abaixo:

```
ronaldo@minerva:~$ wget ftp://ftp.ruby-lang.org/pub/ruby/ruby-1.8.5.tar.gz
ronaldo@minerva:~$ tar xvfz ruby-1.8.5.tar.gz
ronaldo@minerva:~$ cd ruby-1.8.5
ronaldo@minerva:~$ ./configure --prefix=/usr
ronaldo@minerva:~$ make
ronaldo@minerva:~$ make install
```

Para verificar a versão instalada do Ruby, use o seguinte comando:

```
ronaldo@minerva:~$ ruby -v
ruby 1.8.4 (2005-12-24) [i486-linux]
```

No meu caso, estou usando a última versão estável da 1.8.4.

Uma vez que o Ruby esteja instalado, é hora de fazer o mesmo com o Rails. O procedimento é bem simples:

basta executar o comando abaixo¹:

```
ronaldo@minerva:~$ gem install rails --include-dependencies
```

Gem é o gerenciador de pacotes do Ruby, similar em funcionamento ao apt-get do Debian ou ao YaST do Suse, capaz de baixar os arquivos necessários da Web e instalá-los no local apropriado dentro das bibliotecas do Ruby sem que o desenvolvedor tenha que se preocupar com os detalhes do processo. Mais à frente, utilizaremos novamente o comando para instalar outros pacotes que facilitam a vida do desenvolvedor Rails. O último parâmetro acima garante que todas as dependências do Rails sejam instaladas simultaneamente sem necessidade de intervenção.

Novas versões do Rails são instalados por um processo similar. Se você já baixou alguma versão alguma vez, use o mesmo comando acima para atualizar a sua distribuição para a versão mais recente, lembrando, principalmente, de que a versão 1.1.6 corrige um sério problema de segurança nas versões anteriores e deve ser a única usada em um servidor público.

Com os passos acima finalizados e um banco de dados disponível, você está pronto para começar o desenvolvimento em Rails.

Começando a aplicação

PRIMEIROS PASSOS

Vamos começar a trabalhar em uma aplicação bem simples.

Como todo tutorial já produzido na face da terra utiliza uma variação de *blogs*, lojas virtuais, agendas, livros de endereços, ou listas de coisas a fazer, vamos tentar algo diferente: produzir uma aplicação simples usando alguns poucos conceitos do GTD, a metodologia de produtividade criada por David Allen. Vide *Getting Things Done*, por David Allen—traduzido no Brasil como *A Arte de Fazer Acontecer*—para mais detalhes.

Obviamente, como o GTD é uma metodologia complexa em sua totalidade, vamos nos focar somente em alguns aspectos básicos da mesma como contextos, projetos, e próximas ações. Isso será mais do que suficiente para passarmos pela maioria dos componentes do Rails. Sendo o que estamos criando meramente uma implementação ligeira desses conceitos, o resultado final não estará, é claro, polido para o uso diário. Da mesma forma, você não precisa entender os processos do GTD para seguir o tutorial—as partes que vamos utilizar são suficientemente auto-explicativas.

¹ Lembre-se de que, no Linux, você precisará de permissões suficientes para executar o comando acima, possivelmente logando como superusuário—a menos, é claro, que você tenha instalando o Ruby e o Rails em seu próprio diretório *home*.

A primeira coisa a fazer, então, é criar o diretório de trabalho da aplicação. Isso é feito com o comando abaixo:

```
ronaldo@minerva:~/tmp$ rails gtd
create
create app/controllers
create app/helpers
create app/models
create app/views/layouts
create config/environments
create components
create db
create doc
create lib
create lib/tasks
create log
create public/images
create public/javascripts
create public/stylesheets
create script/performance
create script/process
create test/fixtures
create test/functional
create test/integration
create test/mocks/development
create test/mocks/test
create test/unit
create vendor
create vendor/plugins
...
```

O comando `rails` gera o esqueleto completo de uma aplicação, pronta para rodar. Esse comando foi criado em seu sistema quando você instalou as bibliotecas necessárias. Caso você não consiga rodá-lo no Windows, é possível que as suas variáveis de ambiente não foram atualizadas—seja porque a janela de comando em que você está trabalhando já estava aberta ou por algum outro motivo qualquer. Nesse caso, garanta que o diretório `bin` da sua instalação Ruby esteja no caminho do Windows, adicionando-o à sua variável `%PATH%`, reiniciando qualquer sessão de comando conforme o necessário.

No esqueleto gerado, cada parte da aplicação tem um local específico para ser colocado. Isso deriva de uma das filosofias por trás do Rails que pode ser descrita pela frase “convenção ao invés de configuração”. Convenção, nesse caso, significa que há um acordo entre o *framework* e você, o desenvolvedor, de modo que você não precisa de preocupar com certos detalhes que, de outra forma, teriam que se descritos em um arquivo de configuração.

Essa característica, inclusive, gerou uma piada recorrente na comunidade segundo a qual muitas aplicações Rails completas são menores do que um único arquivo de configuração de uma aplicação Java. E considerando o monte de siglas que você precisa conhecer para criar mesmo uma aplicação pequena em qualquer *framework* nessa última linguagem, não é muito difícil acreditar que esse realmente seja o caso.

No Rails, basta executar uma determinada ação ou criar um determinado arquivo e as coisas funcionarão da maneira que você deseja e espera automaticamente, sem necessidade de qualquer trabalho adicional. A princípio isso pode parecer um pouco restritivo, mas na prática funciona perfeitamente. E sendo

extremamente flexível, o Rails permite que você mude qualquer coisa que precise, quando precisar.

Não vamos entrar em detalhes sobre a estrutura de diretórios agora porque ela ficará evidente à medida que avançarmos no tutorial. Não se preocupe: ela é bem lógica e, na maior parte do tempo, automaticamente gerenciada pelo Rails.

Para fins do presente tutorial, vamos seguir a convenção do comando acima e rodar todos os outros que precisarmos diretamente, modificando os arquivos necessários em um editor de texto qualquer. Para o desenvolvimento diário, porém, existem ambientes específicos para o Rails com facilidades próprias para a linguagem Ruby e para os detalhes do *framework*. Falaremos um pouco mais sobre isso depois.

Agora que temos uma aplicação básica, vamos rodá-la e ver o resultado.

Uma das vantagens do Rails é que o ciclo de codificar-rodar-e-testar é bem rápido. Basta fazer uma alteração para ver imediatamente o resultado, sem a necessidade de recarregar processos, esperar a compilação da aplicação ou enviá-la para um servidor especialmente preparado. O máximo que você terá que fazer no Rails é reiniciar o servidor Web no caso de alguma mudança fundamental na configuração básica da aplicação, algo que ocorre com pouca freqüência.

Para facilitar a vida, o Rails vem com seu próprio servidor Web, utilizando as bibliotecas do próprio Ruby. Para rodar o servidor, basta usar um dos *scripts* utilitários presentes em cada aplicação gerada pelo Rails (veja o diretório *script* sob o esqueleto da aplicação).

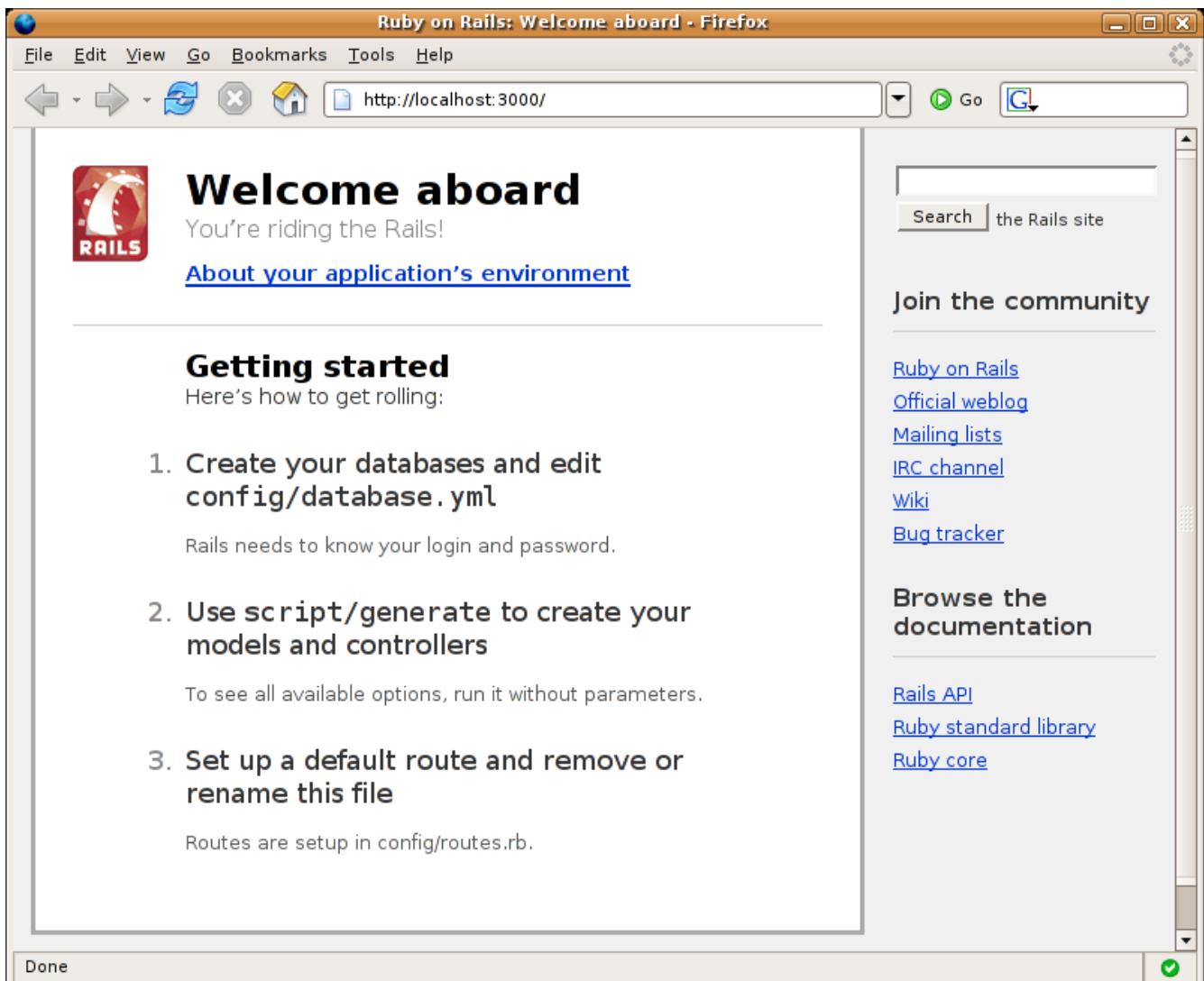
O comando, executado dentro do diretório da aplicação, é o seguinte²:

```
ronaldo@minerva:~/tmp/gtd$ script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2006-09-20 10:31:40] INFO  WEBrick 1.3.1
[2006-09-20 10:31:40] INFO  ruby 1.8.4 (2005-12-24) [i486-linux]
[2006-09-20 10:31:40] INFO  WEBrick::HTTPServer#start: pid=8262 port=3000
```

Como você poder ver, o comando inicia uma instância do servidor WEBrick, capaz de servir aplicações Rails sem necessidade de qualquer configuração adicional. O servidor roda localmente e recebe requisições na porta 3000. Sendo um servidor simples e embutido, o WEBrick não é capaz de receber requisições simultâneas, mas permite que testemos perfeitamente a nossa aplicação.

Acessando o endereço da aplicação você tem o seguinte:

² No Windows, será necessário prefixar o comando com uma invocação ao *ruby*, da seguinte forma: *ruby script/server*. Isso acontece porque, no Windows, o *script* usado não é diretamente executável, a menos, é claro, que você esteja usando um ambiente como o Cygwin. Arquivos com a extensão .rb, porém, serão executáveis, quando a instalação do Ruby tiver sido feita usando o “One-Click Installer”.



Como mostrado acima, temos uma aplicação inicial rodando que, inclusive, mostra quais são os próximos passos para continuar o desenvolvimento da mesma.

CONFIGURANDO O BANCO DE DADOS

Vamos acatar a sugestão da página inicial e criar e configurar o banco de dados da aplicação. Esse primeiro passo é muito importante porque o Rails usa as informações proveniente do *schema* do banco de dados para gerar automaticamente arquivos e configurações adicionais. Esse é mais um exemplo de convenção ao invés de configuração. Dessa forma, garantir que o banco está funcionando corretamente, configurado para uso da aplicação, é o passo inicial de qualquer desenvolvimento em Rails.

O arquivo de configuração de banco de dados se chamada `database.yml` e está localizado no diretório `config`, junto com os demais arquivos de configuração da aplicação.

Esse arquivo, removendo os comentários (as linhas iniciadas com #), tem o seguinte formato:

```
development:
  adapter: mysql
  database: gtd_development
  username: root
  password:
  host: localhost

test:
  adapter: mysql
  database: gtd_test
  username: root
  password:
  host: localhost

production:
  adapter: mysql
  database: gtd_production
  username: root
  password:
  host: localhost
```

A extensão .yml se refere ao formato do arquivo, que utiliza uma linguagem de domínio chamada YAML. Por hora, basta saber que é uma linguagem de serialização de dados favorecida por desenvolvedores Ruby e Rails. Você pode encontrar mais informações sobre a mesma na documentação oficial do Ruby.

Uma aplicação Rails geralmente utiliza três bancos de dados, como demonstrado acima, um para cada ambiente de desenvolvimento padrão.

Um banco de dados é utilizado para o desenvolvimento, onde todas as mudanças são aplicadas. Esse banco tem seu correspondente em um banco de produção, onde modificações somente são aplicadas uma vez que estejam completas. O arquivo permite configurar, inclusive, um banco remoto para onde suas modificações finais serão redirecionadas—embora geralmente seja melhor utilizar um outro método de implantação, como veremos mais adiante.

O terceiro banco mostrado acima é um banco de testes, utilizado pelo Rails para a execução de *unit testing*. Esse banco **deve** ser mantido necessariamente à parte já que todos os dados e tabelas presentes no mesmo são excluídos e recriados a cada teste completo efetuado na aplicação.

Vamos criar o banco agora:

```
ronaldo@minerva:~/tmp/gtd$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 8 to server version: 5.0.22-Debian_0ubuntu6.06.2-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database gtd;
Query OK, 1 row affected (0.56 sec)
```

```
mysql> create database gtd_test;
Query OK, 1 row affected (0.53 sec)

mysql> use gtd;
Database changed

mysql>
```

Modificando o arquivo de configuração para desenvolvimento local, teríamos algo assim:

```
development:
  adapter: mysql
  database: gtd
  username: root
  password:
  host: localhost
  socket: /var/run/mysqld/mysqld.sock

test:
  adapter: mysql
  database: gtd_test
  username: root
  password:
  host: localhost
  socket: /var/run/mysqld/mysqld.sock

production:
  development
```

No caso acima, eu configurei os banco de produção e desenvolvimento para apontarem para o mesmo local já que o desenvolvimento está limitado à minha máquina. Essa é uma estratégia interessante a usar quando queremos testar uma aplicação nas duas situações localmente sem nos preocuparmos em copiar o banco a todo momento—mesmo que o Rails permita isso com pouco mais do que alguns comandos. Eu precisei também acrescentar o método de transporte *socket* à configuração já que em minha máquina, especificamente, o MySQL não permite conexões de rede em sua configuração padrão.

Outros tipos bancos de dados são configurados de maneira similar, mudando o parâmetro *adapter* e quaisquer outras configurações necessárias. Uma maneira mais simples de gerar uma arquivo de configuração específico para o banco de dados que você está usando é já invocar o comando para gerar o esqueleto da aplicação passando um parâmetro para isso. Por exemplo:

```
rails --database=oracle gtd
```

O comando acima gera o mesmo esqueleto da aplicação, com a única diferença de que o arquivo *database.yml* será um pouco diferente para levar em conta as diferenças do Oracle.

Voltando à nossa aplicação, como um arquivo de configuração importante foi mudado, o servidor Web precisa ser reiniciado. Isso acontece porque ele somente lê essas configurações no início de execução. Esse é um dos raros casos em que um servidor de desenvolvimento precisa ser reiniciado já que o Rails recarrega

praticamente qualquer modificação feita na aplicação quando está no modo de desenvolvimento.

Agora temos uma aplicação e um banco de dados configurado. Com isso já podemos iniciar o processo de desenvolvimento propriamente dito.

Para acesso ao banco de dados, o Rails usa classes de dados conhecidas como *models*, ou modelos de dados. Essas classes mascaram os detalhes “sórdidos” do acesso ao banco de dados providenciando uma interface fácil e intuitiva ao desenvolvedor. Em qualquer aplicação, teremos pelo menos um modelo de dados para utilizar na mesma. E a maneira mais fácil de fazer isso é criar automaticamente a tabela no banco de dados, utilizando um dos *scripts* do Rails.

O Rails automaticamente entende que todas as tabelas criadas no banco para uso em modelos, ou classes de dados, satisfarão a duas condições: terão um nome plural em inglês e terão uma chave primária surrogada inteira e auto-incrementada chamada *id*. É possível mudar ambas as condições, mas inicialmente ficaremos com elas para não ter que modificar o que o Rails gera e usa automaticamente.

Para facilitar o desenvolvimento, a aplicação desenvolvida está em inglês, que espero seja simples o suficiente mesmo para a compreensão do leitor que não domine plenamente este idioma. Considerando a profusão de termos em inglês em nossa área acho que isso não será um problema. O uso do inglês evita que tenhamos que pensar em alguns detalhes de tradução e globalização enquanto estamos aprendendo o Rails. Mais no fim do tutorial explicaremos como mudar algumas das coisas que o Rails por como padrão.

GERANDO MIGRAÇÕES E MODELOS DE DADOS

Geralmente um tutorial de Rails começa mostrando com podemos já adicionar algumas ações à interface. Começaremos, entretanto, com a geração de migrações e modelos de dados. Essa é uma opção de desenvolvimento para o presente tutorial, mas você pode seguir em qualquer direção que achar melhor uma vez que esteja desenvolvendo suas próprias aplicações. Isso dito, vamos continuar.

Para manter portabilidade e facilitar o controle de versionamento do banco de dados, o Rails utiliza algo chamado *migrations* (migrações). São *scripts* em Ruby descrevendo modificações de qualquer natureza no banco de dados. Migrações são a maneira mais fácil de acrescentar tabelas e classes de dados ao Rails e é o que utilizaremos agora.

A primeira tabela que vamos criar será a tabela de contextos. Contextos, no GTD, são, basicamente, os ambientes onde uma ação será executada, como, por exemplo, casa, escritório, e-mail, telefone, etc. O nome é bem intuitivo e você pode ler mais sobre o assunto do livro mencionado anteriormente.

Vamos gerar então uma migração para essa primeira modificação no banco:

```
ronaldo@minerva:~/tmp/gtd$ script/generate migration create_contexts
create db/migrate
```

```
create db/migrate/001_create_contexts.rb
```

O `script generate` é uma das ferramentas cruciais do Rails que utilizaremos ao longo de todo nosso tutorial, sendo usado, como o nome indica, para criar basicamente qualquer estrutura que precisamos em uma aplicação. O seu primeiro parâmetro é o tipo de objeto que estamos gerando, seu segundo parâmetro é o nome desse objeto e quaisquer outros parâmetros adicionais indicam opções de geração. No caso acima, estamos gerando uma migração, cujo nome é `create_contexts`.

No Rails, existem muitas convenções quanto ao uso de nomes, que são seguidas pelos comandos do mesmo, com conversões automáticas quando necessário. No caso acima, por exemplo, o nome que usamos será automaticamente convertido em um nome de classe, que no Ruby são expressas com capitalização alternada, como você poderá ver abaixo.

O resultado do comando é um arquivo novo, prefixado pela versão da migração, que foi criado no diretório `db/migrate`. Esse prefixo é usado para controlar quais alterações um banco precisa sofrer para chegar a uma versão específica.

Editando o arquivo, temos o seguinte:

```
class CreateContexts < ActiveRecord::Migration
  def self.up
  end

  def self.down
  end
end
```

O Rails criou uma classe derivada (`<`) da classe `ActiveRecord::Migration` com dois métodos. Esses métodos, por estarem prefixados por `self`, indicam no Ruby que os mesmos podem ser chamados diretamente na classe (métodos estáticos na terminologia do C++, C# e do Java), sem necessidade de uma instância.

O método `self.up` é utilizado para efetuar as modificações. O seu oposto, `self.down`, é usado para desfazer essas modificações caso você esteja revertendo para uma versão anterior do banco. Dependendo da forma como os métodos forem escritos, eles serão realmente complementares. Obviamente é possível que modificações sejam efetuadas que não são reversíveis. Nesse caso, o Rails emitirá um erro se uma migração para uma versão mais baixa for tentada.

Vamos editar o arquivo agora para incluir a tabela que desejamos criar:

```
class CreateContexts < ActiveRecord::Migration
  def self.up
    create_table :contexts do |t|
```

```

    t.column :name, :string
  end
end

def self.down
  drop_table :contexts
end
end

```

Para efetuar suas migrações, o Rails utiliza uma linguagem de domínio que permite especificar operações de banco de dados de forma abstrata, que não está presa a um servidor específico. É possível executar operações diretamente via SQL no banco, mas isso não é recomendado por quebrar essa abstração.

No caso acima, o método `create_table` dentro de `self.up` serve para especificar a tabela que será criada. Esse método recebe um bloco como parâmetro (indicado pela palavra-chave `do`) que especifica as colunas que serão adicionadas.

Blocos são uma das partes mais interessantes do Ruby e vale a estudar um pouco o que eles podem fazer já que o uso dos mesmos é extenso em qualquer aplicação na linguagem, incluindo o Rails. No Ruby, os blocos são valores de primeira classe, que pode ser atribuídos e manipulados como qualquer outro objeto.

Um outro detalhe a manter em mente é o uso extensivo de símbolos no Ruby. No caso acima, o próprio nome da tabela é denotado por um símbolo. Símbolos são similares a *strings*, com a diferença fundamental que são internalizados e existem com uma única instância que permite um uso transparente e eficiente dos mesmos.

Voltando ao nosso código, como a tabela terá somente uma coluna inicialmente, que é o nome do contexto, somente precisamos de uma linha, identificando o nome da coluna e seu tipo. Mais à frente veremos outros tipos e opções. O método `self.down`, como explicando, realiza a operação inversa destruindo a tabela.

Agora é hora de aplicar essa migração ao banco, subindo sua versão. Para isso, temos o utilitário `rake`, que é parte do Ruby. O `rake` é o equivalente Ruby do `make`, sendo capaz de realizar tarefas descritas em um `Rakefile`, da mesma forma que o `make` faz uso de um `Makefile`.

O Rails vem com seu próprio `Rakefile` que permite a execução de várias de suas tarefas necessárias, incluindo `db:migrate`, que aplica as migrações ainda não efetuadas a um banco.

Rodamos o comando da seguinte forma:

```

ronaldo@minerva:~/tmp/gtd$ rake db:migrate
(in /home/ronaldo/tmp/gtd)
== CreateContexts: migrating =====
-- create_table(:contexts)
  -> 1.2089s
== CreateContexts: migrated (1.2092s) =====

```

Se observamos o banco agora, teremos o seguinte *schema*:

```
ronaldo@minerva:~/tmp/gtd$ mysql -u root -p gtd
mysql> show tables;
+-----+
| Tables_in_gtd |
+-----+
| contexts      |
| schema_info   |
+-----+
2 rows in set (0.00 sec)
```

Duas tabelas foram criadas, *contexts* e *schema_info*. A primeira é a que especificamos. A segunda descreve informações do banco para o Rails, para controlar o versionamento do mesmo.

A tabela *contexts* ficou assim:

```
ronaldo@minerva:~/tmp/gtd$ mysql -u root -p
mysql> describe contexts;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+
| id    | int(11)   | NO   | PRI | NULL    | auto_increment |
| name  | varchar(255)| YES  |     | NULL    |              |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Como você pode ver, não é preciso especificar a chave primária surrogada, que é automaticamente criada pelo Rails.

A tabela *schema_info* possui os seguintes dados:

```
ronaldo@minerva:~/tmp/gtd$ mysql -u root -p gtd
mysql> select * from schema_info;
+-----+
| version |
+-----+
|      1 |
+-----+
1 row in set (0.00 sec)
```

Esse número de versão será incrementado ou decrementado dependendo da direção da migração.

A migração também criou o arquivo `db/schema.rb`, contendo uma representação do banco em Ruby, que pode ser usada para recriar a estrutura do mesmo em qualquer banco suportado pelo Rails, usando uma outra tarefa do `rake`.

O arquivo é auto-gerado e não deve ser modificado manualmente. Seu formato pode ser visto abaixo:

```

# This file is autogenerated. Instead of editing this file, please use the
# migrations feature of ActiveRecord to incrementally modify your database, and
# then regenerate this schema definition.

ActiveRecord::Schema.define(:version => 1) do

  create_table "contexts", :force => true do |t|
    t.column "name", :string
  end

end

```

Note apenas que essa representação é limitada às tabelas em si e não inclui chaves estrangeiras, *triggers* ou *stored procedures*.

O próximo passo agora é criar a classe de dados correspondente à tabela. Para isso, usamos o comando abaixo:

```

ronaldo@minerva:~/tmp/gtd$ script/generate model context
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/context.rb
create  test/unit/context_test.rb
create  test/fixtures/contexts.yml
exists  db/migrate
Another migration is already named create_contexts: db/migrate/001_create_contexts.rb

```

O Rails utiliza uma estratégia de desenvolvimento (conhecidas como *patterns* de maneira geral) chamada de **MVC** (*Model-View-Controller*). Essa estratégia separa os componentes da aplicação em partes distintas que não só facilitam o desenvolvimento como também facilitam a manutenção.

O que estamos vendo no momento são os *models*, que correspondem à camada de acesso ao banco de dados, implementada no Rails por um componente denominado *ActiveRecord*. O comando acima gera toda estrutura de suporte ao modelo, incluindo testes e migrações para o mesmo. Como geramos a nossa migração inicial manualmente, o Rails nos informa que já existe uma com o mesmo nome que ele pretendia gerar e que ele está pulando esse passo.

O arquivo responsável pela implementação do modelo está em `app/model/context.rb` e contém apenas o seguinte:

```

class Context < ActiveRecord::Base
end

```

Essas duas linhas, apoiadas pelo Rails, já providenciam uma riqueza de implementação que nos permite recuperar, inserir e atualizar dados no banco, e executar uma série de outras operações complexas sem a necessidade de qualquer comando SQL direto.

Vamos criar agora um novo modelo correspondendo à tabela de projetos. Vamos deixar que o Rails gere a migração dessa vez. O comando é:

```
ronaldo@minerva:~/tmp/gtd$ script/generate model project
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/project.rb
create  test/unit/project_test.rb
create  test/fixtures/projects.yml
exists  db/migrate
create  db/migrate/002_create_projects.rb
```

Você pode ver que o Rails gerou um *template* da migração, e basta editá-la:

```
class CreateProjects < ActiveRecord::Migration
  def self.up
    create_table :projects do |t|
      # t.column :name, :string
    end
  end

  def self.down
    drop_table :projects
  end
end
```

O arquivo final seria, com base no que pensamos até o momento:

```
class CreateProjects < ActiveRecord::Migration
  def self.up
    create_table :projects do |t|
      t.column :name, :string
      t.column :description, :text
    end
  end

  def self.down
    drop_table :projects
  end
end
```

Você não precisa se preocupar em criar todos os campos inicialmente. Você pode sempre usar outra migração para isso.

Rodando o comando `rake` para carregar as migrações mais uma vez, temos o seguinte:

```
ronaldo@minerva:~/tmp/gtd$ rake db:migrate
(in /home/ronaldo/tmp/gtd)
== CreateProjects: migrating =====
-- create_table(:projects)
 -> 0.1863s
== CreateProjects: migrated (0.1865s) =====
```

O resultado final do banco seria:

```
ronaldo@minerva:~/tmp/gtd$ mysql -u root -p gtd
mysql> show tables;
+-----+
| Tables_in_gtd |
+-----+
| contexts      |
| projects      |
| schema_info   |
+-----+
3 rows in set (0.00 sec)

mysql> describe projects;
+-----+-----+-----+-----+-----+-----+
| Field      | Type       | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+-----+
| id         | int(11)    | NO   | PRI | NULL    | auto_increment |
| name       | varchar(255) | YES  |     | NULL    |             |
| description | text        | YES  |     | NULL    |             |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> select * from schema_info;
+-----+
| version |
+-----+
|      2 |
+-----+
1 row in set (0.00 sec)
```

E o arquivo db/schema.rb agora contém:

```
# This file is autogenerated. Instead of editing this file, please use the
# migrations feature of ActiveRecord to incrementally modify your database, and
# then regenerate this schema definition.

ActiveRecord::Schema.define(:version => 2) do

  create_table "contexts", :force => true do |t|
    t.column "name", :string
  end

  create_table "projects", :force => true do |t|
    t.column "name", :string
    t.column "description", :text
  end

end
```

Para terminar, vamos gerar a tabela e o modelo de ações:

```
ronaldo@minerva:~/tmp/gtd$ script/generate model action
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/action.rb
create  test/unit/action_test.rb
create  test/fixtures/actions.yml
exists  db/migrate
create  db/migrate/003_create_actions.rb
```

A migração para o modelo seria:

```
class CreateActions < ActiveRecord::Migration
  def self.up
    create_table :actions do |t|
      t.column :description, :string
      t.column :done, :boolean
      t.column :created_at, :datetime
      t.column :completed_at, :datetime
      t.column :context_id, :integer
      t.column :project_id, :integer
    end
  end

  def self.down
    drop_table :actions
  end
end
```

Com isso, já temos agora o suficiente em nosso banco de dados para trabalhar um pouco em nossa aplicação.

Um recurso muito útil do Rails é o console, que serve tanto para experimentar com as classes de dados como para executar uma série de outras funções úteis de teste e manutenção da aplicação. Para acessar o controle, use o seguinte comando:

```
ronaldo@minerva:~/tmp/gtd$ script/console
Loading development environment.
>>
```

A exemplo de um console de comando do DOS ou do próprio MySQL, o console do Rails permite que você insira um comando e veja seus resultados imediatamente:

Por exemplo:

```
>> Context.create(:name => '@Home')
=> #<Context:0xb74fbb3c @new_record=false, @errors=#<ActiveRecord::Errors:0xb74cf668
@base=#<Context:0xb74fbb3c ...>, @errors={}, @attributes={"name"=>"@Home", "id"=>1}>
```

O comando acima cria e salva um novo contexto, recebendo como parâmetro o nome do mesmo, descrito por seu único atributo. O resultado, visto logo depois do comando, é uma representação textual da classe.

Todos objetos em Ruby, e isso inclui desde números inteiros a classes complexas, possuem uma representação textual. Na maior parte dos casos, o Ruby gera uma representação textual automática para classes que não especificam a sua própria representação e é isso o que você está vendo acima.

Em uma aplicação, você provavelmente precisará criar um objeto e manipulá-lo antes de salvá-lo. Isso

também é possível, como demonstrado abaixo:

```
>> context = Context.new
=> #<Context:0xb7781bd0 @new_record=true, @attributes={"name"=>nil}>

>> context.name = '@Work'
=> "@Work"

>> context.save
=> true
```

No exemplo acima, o objeto é criado, uma de suas propriedade é atribuída e depois disso o objeto é salvo. A persistência de dados só ocorre na invocação do método `save`.

Outras operações são possíveis:

```
>> Context.find(:all)
=> [#<Context:0xb76e76d4 @attributes={"name"=>"@Home", "id"=>"1"}, #<Context:0xb76e7698
@attributes={"name"=>"@Work", "id"=>"2"}]

>> Context.find(2)
=> #<Context:0xb76e2328 @attributes={"name"=>"@Work", "id"=>"2"}>

>> Context.find(:first, :conditions => ['name like ?', '@Home'])
=> #<Context:0xb76d6bcc @attributes={"name"=>"@Home", "id"=>"1"}>
```

A primeira chamada acima retorna todos os registros correspondente aos contextos. Vemos que o resultado é um *array*, delimitado por `[]`. A segunda chamada, por sua vez, retorna o objeto especificado pelo *id* passado, ou seja, por sua chave primária surrogada. O terceiro método, por fim, retorna o primeiro registro satisfazendo as condições passadas.

O método `find` possui vários outros parâmetros que servem para gerar *queries* complexas sem esforço, ordenando o resultado, limitando a quantidade de itens retornados, e, para usos mais avançados, criando *joins* automaticamente. Veremos exemplos extensos disso mais adiante.

A sofisticação das classes geradas pelo *ActiveRecord* se estende ao ponto da criação de métodos automáticos para várias situações. Algumas delas exploraremos depois, mas vale a pena notar aqui pelo menos uma delas: métodos de busca e criação automáticos.

Por exemplo, é possível executar o comando abaixo para encontrar um contexto pelo seu nome:

```
>> Context.find_by_name('@Home')
=> #<Context:0xb76cdfb8 @attributes={"name"=>"@Home", "id"=>"1"}>
```

O método `find_by_name` não existia até ser chamado. Isso pode ser visto verificando a existência do método, usando `respond_to?`, que é um método presente em todas as classes Ruby e, ao receber um símbolo que especifica o nome do método a ser testado, devolve uma valor indicando se a classe suporta o método ou

não:

```
>> Context.respond_to?(:find_by_name)
=> false

>> Context.find_by_name('@Home')
=> #<Context:0xb796f0c4 @attributes={"name"=>"@Home", "id"=>"1"}>
```

Como é fácil perceber, a classe não possui aquele método até que ele seja invocado. Uma classe de dados Rails aceita combinações quaisquer de seus atributos e de algumas operações para gerar esse tipo de métodos.

Um exemplo é `find_or_create_by_name`, que procuraria um registro e o criaria caso não fosse encontrado, com o nome passado como parâmetro. Se a classe possuir mais atributos, poderíamos muito bem usar algo como `find_or_create_by_name_and_description` ou ainda `find_by_description_and_completed_at`.

É claro que se usarmos isso para todas as chamadas, teremos métodos com nomes absurdamente longos como `find_or_create_by_description_and_created_at_and_completed_at_and_done` que ficariam bem melhor como chamadas separadas usando condições por questões de legibilidade.

O que estamos fazendo aqui no console é basicamente o que faremos em uma aplicação no que tange à manipulação de dados. Como você pode ver, na maior parte dos casos, o Rails não exige que o desenvolvedor escreva código SQL, gerando o código necessário por trás das cenas, com toda eficiência possível. E caso seja necessário, há ainda duas possibilidades: usar parâmetros mais avançados do método `find`, que permitem a inserção de fragmentos de SQL em uma operação qualquer; ou ainda, usar métodos como `find_by_sql` que permitem um execução direta no banco com integração plena com o Rails.

Usar as facilidades do Rails não significa que o desenvolvedor não precisa conhecer SQL; ao contrário, o conhecimento é fundamental, tanto para evitar problemas no uso do próprio Rails como para todas outras tarefas que existem além do Rails.

Agora que já trabalhamos um pouco com o modelo de dados, vamos passar para a aplicação em si.

MVC

Como mencionado anteriormente, o Rails utilizado a estratégia MVC. A parte correspondente ao *Model* existe nas classes de dados e é implementada pelo *ActiveRecord*. Um segundo componente do Rails, o *ActionPack*, implementa o V e o C que são respectivamente *View* e *Controller*.

Um *controller* é uma classe responsável por receber as requisições feitas pela aplicação e executar as ações necessárias para atender essas requisições. Essas ações, ao serem executadas, provavelmente causarão mudanças no banco que serão efetuadas, por sua vez, por uma ou mais classes de dados.

Finalmente, o resultado será exibido através de uma *view*, que será retornada ao usuário na forma de HTML, imagens, XML, ou qualquer outra saída aceitável da aplicação.

As três partes da estratégia MVC são independentes no Rails como deveriam ser em qualquer boa implementação das mesmas. Qualquer modelo de dados pode ser utilizado independentemente, inclusive em *scripts* que não tem a ver com a aplicação Web em si, como, por exemplo, tarefas agendadas. Os *controllers* também são independentes e podem ser usado para simplesmente efetuar ações que nem mesmo retornam *views*, atualizando somente o banco ou disparando outro processo qualquer no servidor. As *views* são, de certa forma, a parte mais dependente já que, embora podendo ser utilizadas separadamente, não fazem muito sentido sem utilizarem dados gerados por modelos de dados e *controllers*.

Para identificar qual *controller* será responsável por atender uma determinada solicitação da aplicação, o Rails utiliza uma mecanismo de roteamento de requisições automático que não necessita de configurações complexas, mas que pode também ser customizado de acordo com as necessidades de cada aplicação.

A regra principal de roteamento no Rails forma URLs segundo o padrão `/controller/action/id`, onde *controller* é a classe responsável por atender a requisição especificada por aquela URL, *action* é um método dentro da classe e *id* é um parâmetro opcional passado para identificar um objeto qualquer sobre o qual a ação será efetuada. Como dito anteriormente, esse padrão pode ser modificado e veremos mais sobre isso adiante no tutorial.

O PRIMEIRO CONTROLLER

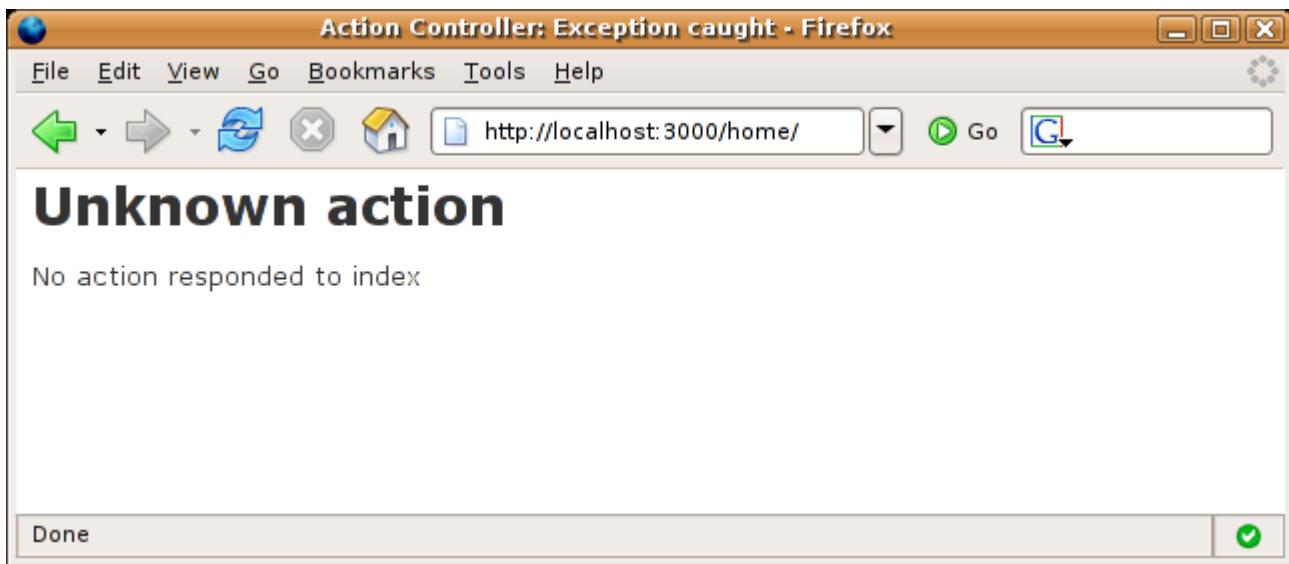
Vamos começar criando um *controller* para lidar com a página inicial de nossa aplicação. Para gerar um *controller*, usamos o comando abaixo:

```
ronaldo@minerva:~/tmp/gtd$ script/generate controller home
exists  app/controllers/
exists  app/helpers/
create  app/views/home
exists  test/functional/
create  app/controllers/home_controller.rb
create  test/functional/home_controller_test.rb
create  app/helpers/home_helper.rb
```

O nome passado para o comando é `home`, que será usado para compor todos os arquivos gerados pelo mesmo.

Da mesma forma que nos modelos de dados, o Rails cria *unit tests* para a classe gerada, que podem ser executados com o comando `rake tests`. Esse comando roda todos os *unit tests* presentes na aplicação e qualquer boa aplicação desenvolvida segundo a metodologia *Extreme Programming* faz uso extensivo dessa técnica. Veremos esse assunto em mais detalhes em uma outra seção de nosso tutorial.

Se você acessar a URL desse *controller* agora, você vai receber a seguinte tela:



Repare que, nesse caso, eu informei somente `/controller/` como a URL, sem passar nem a parte correspondente a uma ação ou a um *id*. Nesse caso, o Rails assume que estamos invocando a ação *index* sem *id*. Esse é um mais exemplo de convenção ao invés de configuração. Ao invés de ter sempre que especificar uma ação padrão em algum lugar, o Rails convenciona que a ação padrão é *index*, poupano o desenvolvedor sem afetar a aplicação.

Vamos olhar o arquivo `app/controllers/home_controller.rb` gerado por nosso comando. Como já mencionamos, o Rails segue uma estrutura fixa de diretórios, poupano mais uma vez o desenvolvedor. Arquivos relacionados ao banco vão em no diretório `db`, arquivos de configuração em `config` e tudo relacionado a MVC vai em `app`. Dentro de `app` temos vários outros diretórios que contém mais partes específicas da aplicação. Em alguns casos, o Rails também adiciona um sufixo ao arquivo, evitando colisões de nomes e problemas estranhos na aplicação, como é o caso aqui. O arquivo do *controller* criado contém o seguinte:

```
class HomeController < ApplicationController
end
```

A classe `HomeController` define quem que irá responder a requisições padrão em `/home`. Ela herda da classe `ApplicationController`, que está definida no arquivo `application.rb`, no mesmo diretório. Sendo assim, qualquer método criado na classe `ApplicationController` estará automaticamente disponível nos *controllers* gerados para a aplicação.

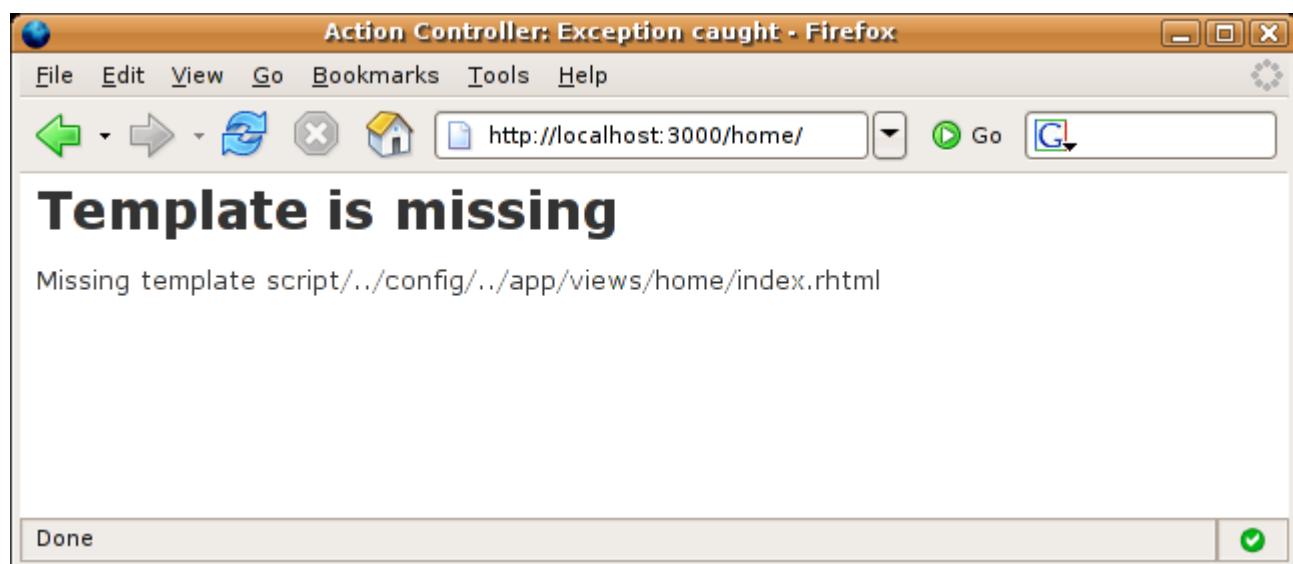
O uso de herança serve mais uma vez para beneficiar o desenvolvedor que pode utilizar um modelo mental familiar para trabalhar sua aplicação, um modelo que é ao mesmo tempo simples e poderoso.

Para criarmos a ação *index*, basta adicionarmos um método à classe:

```
class HomeController < ApplicationController  
  def index  
  end  
end
```

O princípio que usamos acima é comum a tudo em Rails. Basicamente tudo o que fazemos em uma aplicação usando o mesmo consiste em extender alguma classe por meio da adição de métodos customizados.

Recarregando a página no navegador, ficamos com o seguinte:



Vamos que, dessa vez, o Rails identificou a ação a ser executada, mas, como a aplicação não especificou nenhum retorno, houve um erro. Isso aconteceu porque o Rails tentou aplicar automaticamente uma *view* para aquela ação do *controller*. Como a *view* ainda não existe, temos o erro.

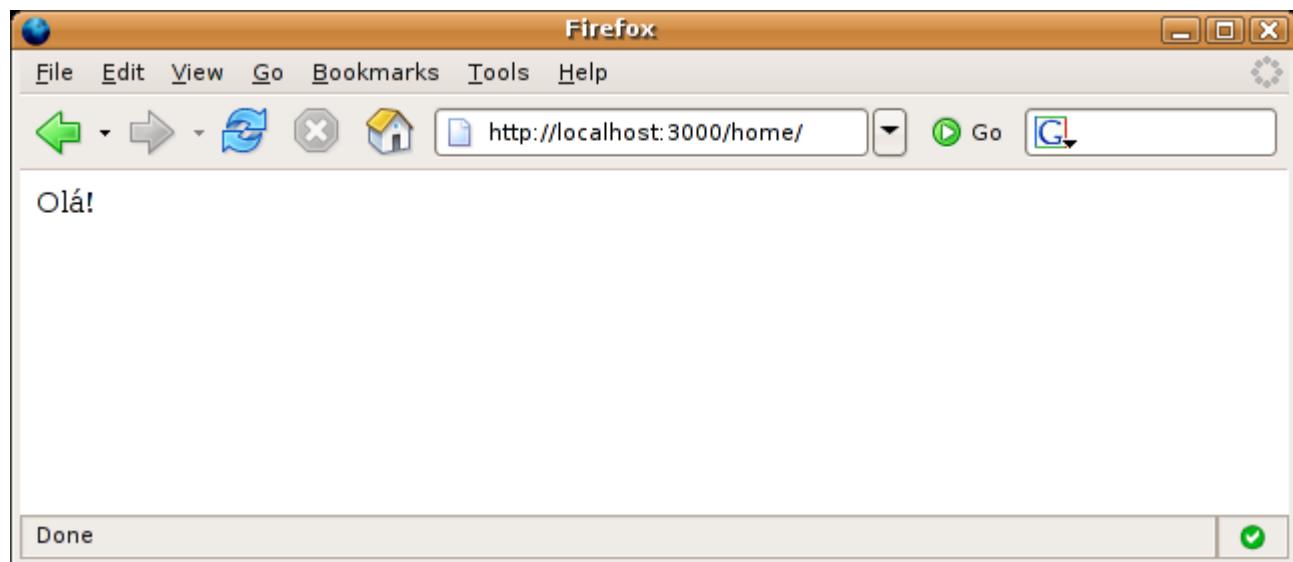
Antes de nos aventurarmos em uma *view* usando o mecanismo de *templates* do Rails, vamos retornar algum texto por conta própria:

```
class HomeController < ApplicationController  
  def index  
    render :text => "Olá!"  
  end  
end
```

O método *render*, disponível para qualquer *controller* ou *view*, gera saída na aplicação, saída esta que depende de seus parâmetros. No caso acima, estamos renderizando texto puro, como indicado pelo

parâmetro `text`.

Nossa página agora ficaria assim:



Temos a nossa primeira saída em uma aplicação Rails.

Como escrever manualmente toda a saída não é o que queremos, vamos criar a nossa primeira *view*. Para isso, criamos o arquivo `app/views/home/index.rhtml`. A extensão `.rhtml` indica que esse arquivo contém HTML e código Ruby.

Uma outra extensão possível seria `.rjs` para gerar retorno em JavaScript, comumente utilizado em aplicações Ajax. Há ainda `.rxml`, para gerar saída em XML. Outros mecanismos de *template* podem usar outras extensões, já que o Rails pode ser configurado para outros mecanismos alternativos.

Por hora, usaremos a linguagem de *templates* padrão do Rails. Vamos criar o seguinte arquivo, então:

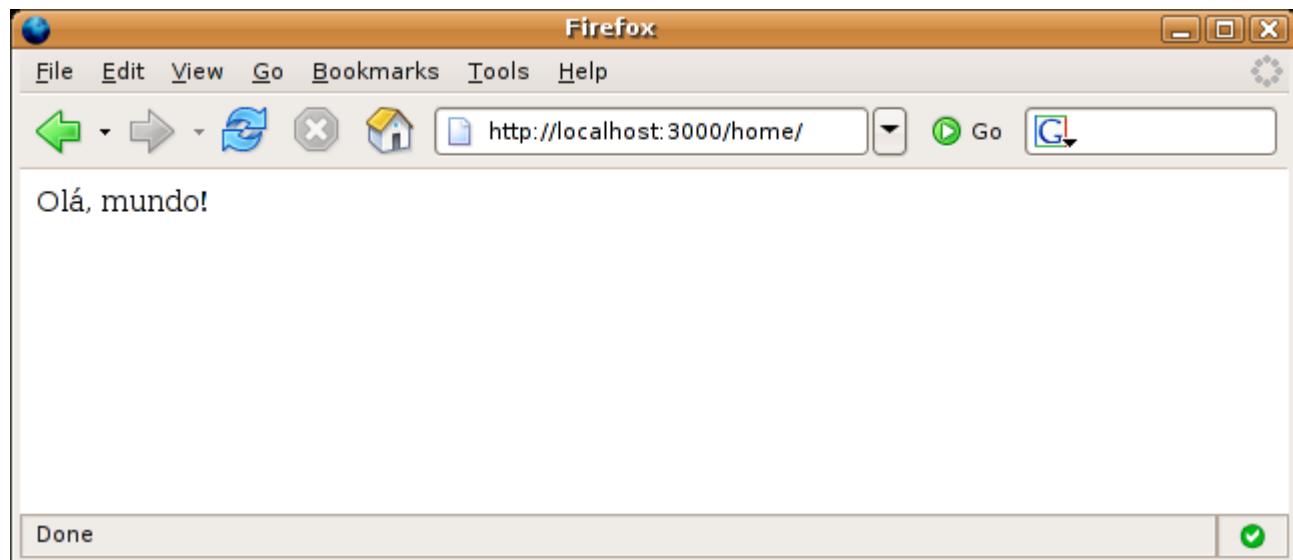
```
<p>Olá, mundo!</p>
```

Se recarregamos a página, notaremos que nada mudou. Isso acontece porque estamos usando `render` diretamente. Nesse caso, o Rails detecta que alguma saída já foi gerada e não tenta gerar outra. Basta, então, remover a chamada a `render` método `index`, voltando o arquivo a:

```
class HomeController < ApplicationController  
  def index  
  end
```

```
end
```

Agora, recarregando a página, temos:



Sem que haja necessidade de especificar qualquer coisa, o Rails está usando o arquivo adequado. Veja que não configuramos qualquer coisa. Criamos um *controller*, definimos um método no mesmo, e criamos um arquivo que contém o que queremos que seja retornado por aquele método. A simples existência desses arquivos representa uma cadeia de execução sem que precisamos nos preocupar com que parte da aplicação faz isso ou aquilo.

Qualquer outra ação que fosse criada dentro desse *controller* seguiria o mesmo padrão. O nome da ação seria associado a um nome de arquivo dentro de um diretório em `app/views` cujo nome seria o do próprio *controller*. O Rails também é inteligente ao ponto de responder a uma ação mesmo que o método não exista, desde que a *view* esteja presente no diretório correto.

LAYOUTS

Para facilitar o desenvolvimento da parte visual de uma aplicação, o Rails possui um conceito denominado *layouts*.

Na maioria das aplicações Web, as páginas variam somente no seu conteúdo principal, possuindo cabeçalhos, rodapés e barras de navegação em comum. Obviamente, um *framework* cujo maior objetivo é aumentar a produtividade do desenvolvedor não exigiria que o código para esses elementos tivesse que ser repetido em cada *view*. Um *layout* funciona como um arquivo raiz, dentro do qual o resultado de uma *view* é inserido automaticamente.

Mais uma vez favorecendo convenção ao invés de configuração, o Rails define um arquivo padrão de *layout*

que é usado automaticamente por qualquer *view* a não ser que haja especificação em contrário. O Rails também é inteligente o bastante para somente usar um *layout* em *views* com a mesma extensão.

O *layout* padrão para a aplicação fica em um arquivo chamado `application.rhtml`, dentro do diretório `app/views/layouts`, que não existe ainda.

Se você olhar agora o código gerado pela página que criamos até o momento, você verá o seguinte:



```
<p>Olá, mundo!</p>
```

Como você pode notar, ao estarmos gerando nenhum dos elementos HTML geralmente vistos em uma página comum.

Vamos criar o arquivo `application.rhtml` no diretório especificado, com o seguinte conteúdo:

```
<html>
<head>
  <title>GTD</title>
</head>

<body>
  <%= yield %>
</body>

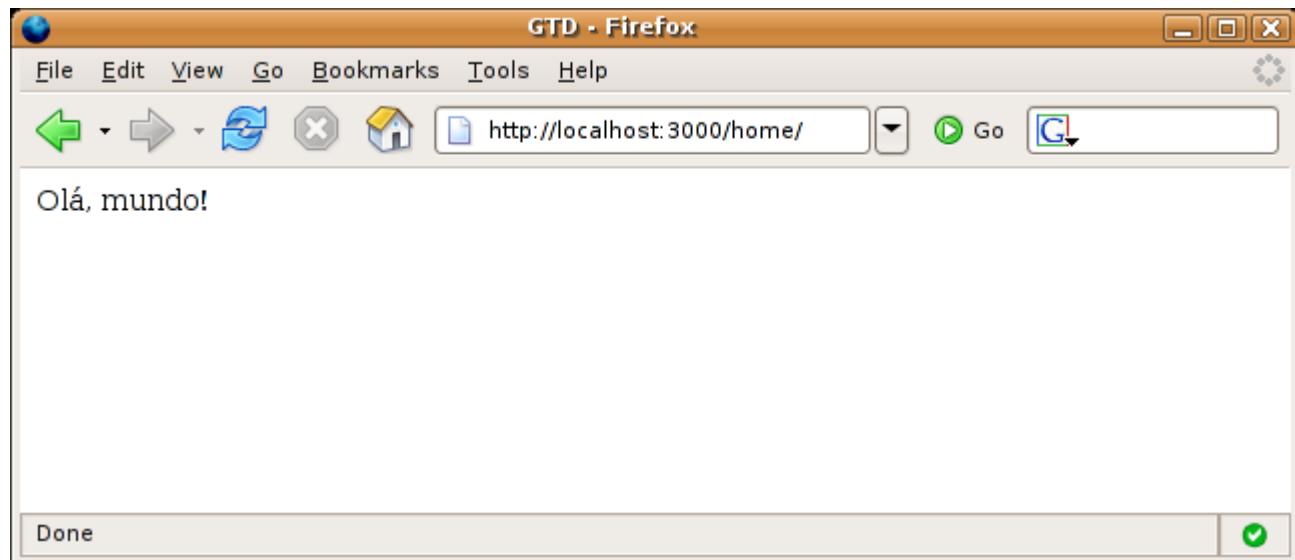
</html>
```

Temos no arquivo acima, o primeiro exemplo de uso de código Ruby dentro de uma *view*, delimitado pelos marcadores `<%` e `%>`. Aqueles familiarizados com PHP e ASP reconhecerão o estilo de marcadores, com o uso de `<%= objeto %>` para retornar conteúdo.

No caso acima, o método especial `yield` retorna o conteúdo atual gerado pela ação, seja por meio de uma *view* ou usando `render` diretamente. O método `yield` tem uma conotação especial no Ruby, servindo para

invocar o bloco associado ao contexto. Inserido em um *layout* do Rails, o bloco define a execução da ação, com seu consequente retorno de conteúdo.

A nossa página recarregada agora fica como mostrado abaixo:



Não parece muita coisa, mas, olhando o código, você verá o seguinte:

```
<html>
<head>
    <title>GTD</title>
</head>

<body>
    <p>Olá, mundo!</p>
</body>

</html>
```

É fácil perceber que o código do *layout* foi aplicado sobre o código da *view*, gerando a saída final da aplicação.

O que precisamos fazer agora é extender o nosso *layout* para incluir algumas amenidades. O nosso arquivo `application.rhtml` poderia ser mudado para o seguinte:

```

<html>
<head>
  <title>GTD</title>
  <%= stylesheet_link_tag "default" %>
</head>

<body>
  <%= yield %>
</body>

</html>

```

O método `stylesheet_link_tag` recebe o nome de uma *stylesheet* como parâmetro e gera um link para a mesma. O nosso código gerado agora ficou assim:

```

<html>
<head>
  <title>GTD</title>
  <link href="/stylesheets/default.css?" media="screen"
rel="Stylesheet" type="text/css" />
</head>

<body>
  <p>Olá, mundo!</p>
</body>

</html>

```

Note que mais uma vez o Rails assumiu um caminho padrão. Nesse caso, o arquivo é servido diretamente da raiz da aplicação, que é o diretório `public`. Você pode criar um arquivo chamado `default.css` no diretório `public/stylesheets` para adicioná-lo à aplicação. Um exemplo disso seria:

```

body
{
  font-family: Verdana, Arial, sans-serif;
  font-size: 80%;
}

```

Uma coisa a manter em mente é que o *layout* padrão da aplicação não é, de forma alguma, o único que pode ser gerado. Você pode criar tantos *layouts* quanto precisar, colocando-os no mesmo diretório, de onde estarão acessíveis a toda aplicação.

Para usar um *layout* diferente em um controler você poderia fazer algo assim:

```

class HomeController < ApplicationController

  layout "home"

  def index
  end

end

```

O *layout* cujo nome é *home* seria especificado no arquivo `app/views/layouts/home.rhtml`, com a mesma funcionalidade do arquivo `application.rhtml`.

Uma outra possibilidade que o Rails oferece é sobrescrever um *layout* para uma única ação, diretamente no método `render`. Um exemplo seria:

```

class HomeController < ApplicationController

  layout "home"

  def index
  end

  def index_special
    render :action => "list", :layout => "home_special"
  end

end

```

Você pode também suprimir inteiramente um *layout* usando `:layout => false`.

Finalmente, você pode usar maneiras específicas de determinar o *layout* de uma página. Por exemplo:

```

class HomeController < ApplicationController

  layout :choose_layout

  def index
  end

  protected

  def choose_layout
    (current_user.admin?) ? "administration" : "normal"
  end

end

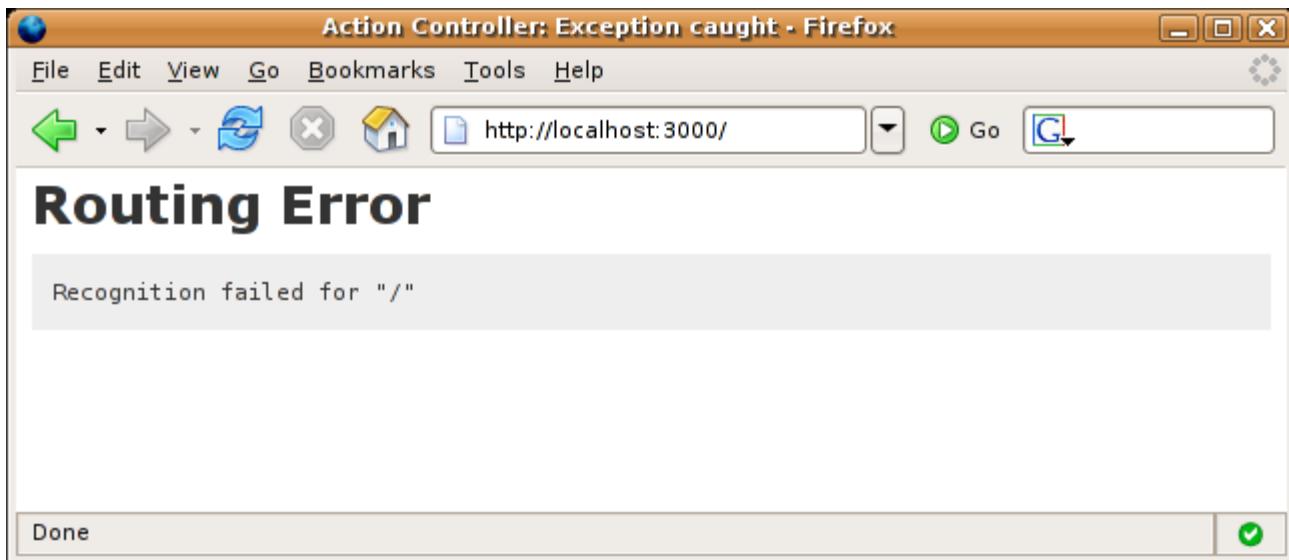
```

No caso acima, dependendo do retorno da chamada ao método `admin?`, a aplicação poderia usar o layout definido no arquivo `administration.rhtml` ou no arquivo `normal.rhtml`. Como podemos ver não há muitos limites para o que pode ser feito.

ROTEAMENTO

Se você acessou a URL raiz da aplicação, você terá notado que ela não mudou, apesar do *controller* que criamos. Isso acontece porque o Rails define um arquivo `index.html` que serve como padrão.

Se removermos esse arquivo do diretório `public`, ficaremos com a seguinte página:



Esse erro indica que o Rails não consegue encontrar um roteamento que satisfaça a URL que especificamos. Para resolvemos o problema, vamos editar o arquivo `config/routes.rb`, que define esses roteamentos. Esse arquivo contém o seguinte código:

```
ActionController::Routing::Routes.draw do |map|
  # The priority is based upon order of creation: first created -> highest priority.

  # Sample of regular route:
  # map.connect 'products/:id', :controller => 'catalog', :action => 'view'
  # Keep in mind you can assign values other than :controller and :action

  # Sample of named route:
  # map.purchase 'products/:id/purchase', :controller => 'catalog', :action => 'purchase'
  # This route can be invoked with purchase_url(:id => product.id)

  # You can have the root of your site routed by hooking up ''
  # -- just remember to delete public/index.html.
  # map.connect '', :controller => "welcome"

  # Allow downloading Web Service WSDL as a file with an extension
  # instead of a file named 'wsdl'
  map.connect ':controller/service.wsdl', :action => 'wsdl'

  # Install the default route as the lowest priority.
  map.connect ':controller/:action/:id'
end
```

Como os comentários dizem, esse arquivo define roteamentos de URLs para *controllers*. Cada roteamento é

examinado quando uma URL é processada pelo Rails, da primeira para a última, sendo a ordem de declaração a definição de prioridade.

Como você pode ver no arquivo acima, a URL padrão que mencionamos, `/controller/action/id` é a última a ser definida, sendo a aplicada caso não haja instruções em contrário. O roteamento no Rails é bem rico e trabalharemos alguns dos conceitos do mesmo mais adiante. No momento, queremos apenas resolver o problema da página inicial.

Essa página é representada no arquivo acima pela linha abaixo, comentada no momento.

```
# map.connect '', :controller => "welcome"
```

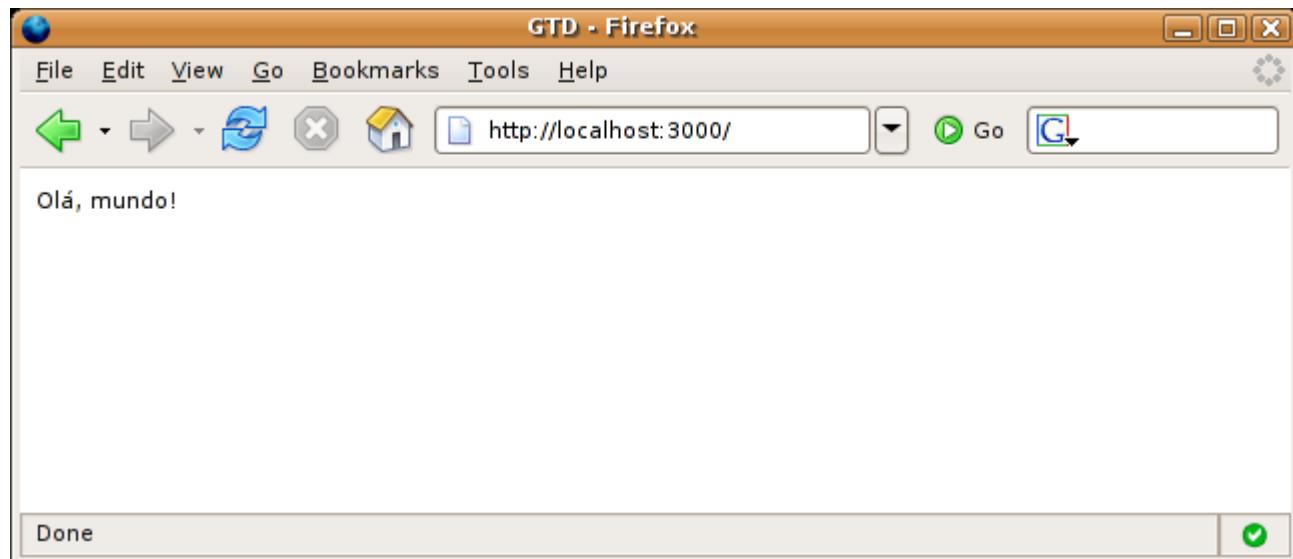
Para mapear o `controller` home como nossa página inicial, basta modificar a linha para dizer o seguinte:

```
map.connect '', :controller => "home"
```

A rota vazia é um sinônimo para a rota raiz e estamos dizendo, com o código acima que se a mesma for invocada, a requisição deve ser servida pelo `controller` que definimos anteriormente.

Note a mistura de aspas simples e aspas duplas nesse arquivo. No Ruby, elas são relativamente intercambiáveis e são apenas duas das várias formas de representar texto nessa linguagem. A diferença é que as aspas duplas permitem a interpolação de variáveis como veremos mais adiante no tutorial. Eu tendo a favorecer o uso de aspas duplas, nas isso é uma opção pessoal.

Agora, recarregando nossa página inicial temos:



Note que agora o mesmo *controller* serve dois roteamentos. Essa é apenas uma das possibilidades mais simples que estão disponíveis com o roteamento do Rails.

Agora que temos um conhecimento básico de *models*, *controllers* e *views*, podemos combinar os três.

SCAFFOLDING

Para ajudar na rápida prototipação de aplicações, o Rails possui algo chamado de *scaffolding*.

O *scaffolding* provê uma estrutura básica para operações CRUD (*Create, Retrieve, Update and Delete*), ou seja, aquelas operações básicas que temos na manipulação de dados, gerando interfaces rápidas que podem apoiar o desenvolvimento até que você insira a codificação necessária.

Para experimentar com isso e expandir um pouco nossa aplicação, vamos criar um novo *controller*, que servirá para administrar os nossos contextos:

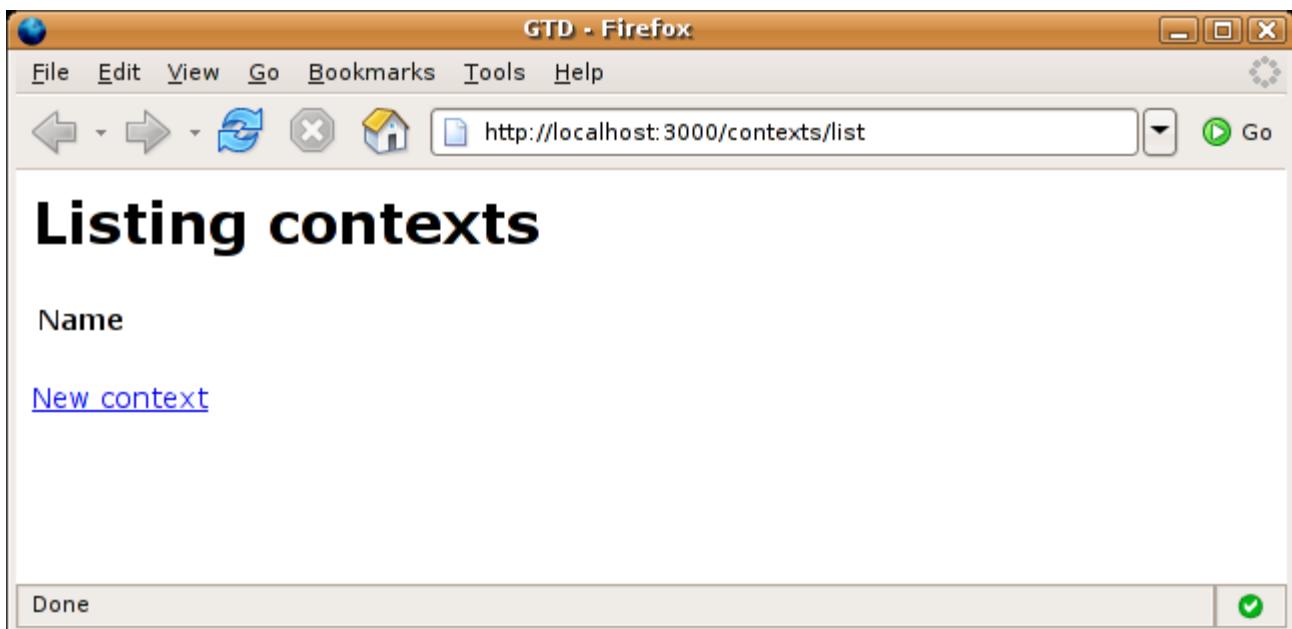
```
ronaldo@minerva:~/tmp/gtd$ script/generate controller contexts
exists app/controllers/
exists app/helpers/
create app/viewscontexts
exists test/functional/
create app/controllerscontexts_controller.rb
create test/functionalcontexts_controller_test.rb
create app/helperscontexts_helper.rb
```

Inicialmente, esse *controller* é como o que criamos anteriormente, para a *home* da aplicação, e não possui nenhuma ação pré-definida.

Vamos editar o seu arquivo, em `app/controllers/context_controller.rb`, acrescentando uma linha de código:

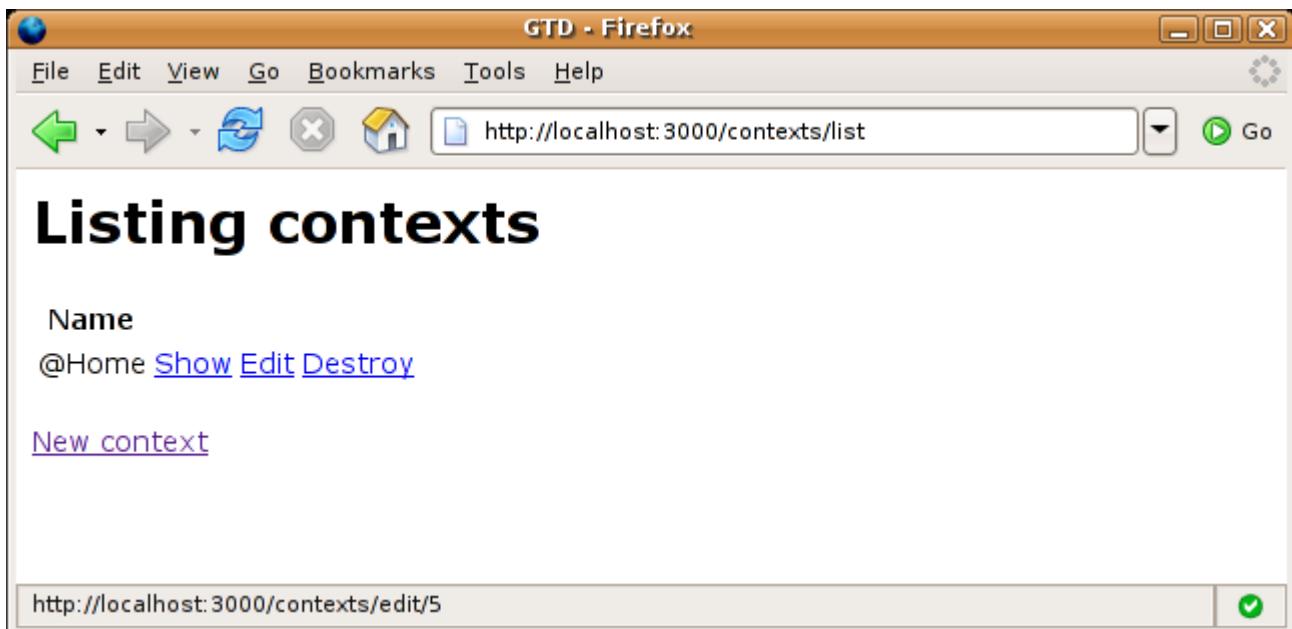
```
class ContextsController < ApplicationController
  scaffold :context
end
```

Agora, acesse a URL desse *controller*, digitando <http://localhost:3000/context> no navegador que você estiver usando:



Sem nenhuma linha de código além da inserida acima, temos listagem, inserção, atualização e remoção de contextos.

Clique em *New context* para experimentar:



Depois, clicando em *Create*, temos:

GTD - Firefox

Name
@Home [Show](#) [Edit](#) [Destroy](#)

[New context](#)

http://localhost:3000/contexts/edit/5

E, finalmente, clicando em *Show*, vemos:

GTD - Firefox

Name: @Home

[Edit](#) | [Back](#)

http://localhost:3000/contexts/destroy/5

Note, que na tela acima, o *id* do objeto já aparece na URL. Nesse caso é 5 porque eu já experimentara antes com outros objetos.

O *scaffolding* é muito útil na geração de *controllers* básicos de administração para dar apoio à prototipação e ao início do desenvolvimento, principalmente para cadastros, permitindo que o desenvolvedor se foque no que é mais importante para a aplicação. O inconveniente é que as ações geradas são bem limitadas e não podem ser editadas ou traduzidas.

Para expandir um pouco a funcionalidade, existe outra forma de geração de *scaffolding* que permite uma flexibilidade maior na edição das páginas geradas, que podem, inclusive, ser utilizadas como base para as páginas finais.

Para isso, use o comando abaixo:

```
ronaldo@minerva:~/tmp/gtd$ script/generate scaffold context
exists  app/controllers/
exists  app/helpers/
exists  app/views-contexts
exists  test/functional/
dependency model
exists    app/models/
exists    test/unit/
exists    test/fixtures/
identical  app/models/context.rb
identical  test/unit/context_test.rb
identical  test/fixtures/contexts.yml
create   app/views-contexts/_form.rhtml
create   app/views-contexts/list.rhtml
create   app/views-contexts/show.rhtml
create   app/views-contexts/new.rhtml
create   app/views-contexts/edit.rhtml
overwrite app/controllers-contexts_controller.rb? [Ynaq] y
  force  app/controllers-contexts_controller.rb
overwrite test/functional-contexts_controller_test.rb? [Ynaq] y
  force  test/functional-contexts_controller_test.rb
identical  app/helpers-contexts_helper.rb
create   app/views/layouts-contexts.rhtml
create   public/stylesheets/scaffold.css
```

Note que, para a geração de um *scaffold* por esse comando, o nome do modelo de dados é que deve ser passado como parâmetro, e não o nome do *controller*. O Rails é capaz de derivar um do outro.

Esse comando gera a classe do modelo de dados, o *controller* e *views* para cada ação CRUD necessária. No caso acima, como o *model* já existia, ele não foi criado, e foi necessário sobrescrever alguns arquivos.

Há duas coisas a serem notadas aqui. Primeiro, o *scaffold* gerou o seu próprio *layout*, com sua própria *stylesheet*. Por causa disso, o layout original da aplicação foi perdido. Uma solução aqui será remover o arquivo *app/views/layouts/contexts.rhtml*, já que não precisamos dele, e adicionar a *stylesheet* *scaffold.css* ao arquivo *application.rhtml*. Segundo, as *views* criadas continuam razoavelmente limitadas, como veremos adiante.

Abrindo o novo arquivo gerado do *controller*, temos o seguinte:

```
class ContextsController < ApplicationController
  def index
    list
    render :action => 'list'
  end
```

```

# GETs should be safe (see http://www.w3.org/2001/tag/doc/whenToUseGet.html)
verify :method => :post, :only => [ :destroy, :create, :update ],
      :redirect_to => { :action => :list }

def list
  @context_pages, @contexts = paginate :contexts, :per_page => 10
end

def show
  @context = Context.find(params[:id])
end

def new
  @context = Context.new
end

def create
  @context = Context.new(params[:context])
  if @context.save
    flash[:notice] = 'Context was successfully created.'
    redirect_to :action => 'list'
  else
    render :action => 'new'
  end
end

def edit
  @context = Context.find(params[:id])
end

def update
  @context = Context.find(params[:id])
  if @context.update_attributes(params[:context])
    flash[:notice] = 'Context was successfully updated.'
    redirect_to :action => 'show', :id => @context
  else
    render :action => 'edit'
  end
end

def destroy
  Context.find(params[:id]).destroy
  redirect_to :action => 'list'
end
end

```

O resultado final, ao ser executado, é muito parecido com o do método `scaffold`, com a maior diferença sendo os arquivos de gerados enquanto que, do modo anterior, as ações eram dinamicamente criadas pelo Rails. Algumas amenidades, como paginação, também foram introduzidas.

Uma consideração deve ser feita aqui: o *scaffolding* é apenas um apoio à prototipação. Ele não deve ser usado como uma muleta, gerando páginas para modificação. Se você quer administrações prontas, existem *plugins* para o Rails que permitem a geração de páginas bem avançadas. Esses *plugins* podem ser utilizados para criar administrações prontas a exemplo das geradas pelo Django (o *framework* equivalente ao Rails para o Python), mas mesmo elas tem suas limitações e inflexibilidades. A lição é sempre procurar o que é melhor para a sua aplicação. Os *scaffoldings* são uma visão muito linear de uma aplicação e não levam em contra interfaces alternativas de melhor usabilidade.

De qualquer forma, o *scaffolding* gerado é uma boa oportunidade para ver o relacionamento entre *controllers*, *views* e *models*. Se você tomar uma ação qualquer como exemplo, verá que ela usa a classe de

dados para o acesso ao banco e empacota os dados recebidos em variáveis fechadas que serão usadas por uma *view*, evitando que o código de apresentação se misture com a lógica de negócio. Mesmo elementos do próprio *controller*, como paginação, são incluídos nessa discriminação. Veja a *view* de listagem, por exemplo, descrita no arquivo `list.rhtml` no diretório `app/views/contexts`:

```
<h1>Listing contexts</h1>

<table>
  <tr>
    <% for column in Context.content_columns %>
      <th><%= column.human_name %></th>
    <% end %>
  </tr>

  <% for context in @contexts %>
    <tr>
      <% for column in Context.content_columns %>
        <td><%= h context.send(column.name) %></td>
      <% end %>
      <td><%= link_to 'Show', :action => 'show', :id => context %></td>
      <td><%= link_to 'Edit', :action => 'edit', :id => context %></td>
      <td><%= link_to 'Destroy', { :action => 'destroy', :id => context }, :confirm => 'Are you sure?', :post => true %></td>
    </tr>
  <% end %>
</table>

<%= link_to 'Previous page', { :page => @context_pages.current.previous } if
@context_pages.current.previous %>
<%= link_to 'Next page', { :page => @context_pages.current.next } if @context_pages.current.next %>

<br />

<%= link_to 'New context', :action => 'new' %>
```

O método `paginate`, usado no *controller* esconde a complexidade de acesso ao banco fazendo a busca dos dados, dividindo os itens retornados pelo número de itens requeridos por página e retornando duas variáveis que contém, respectivamente, uma lista das páginas (que pode ser usada na *view* para gerar a numeração das mesmas) e os itens referentes àquela página. Se você cadastrar mais de 10 itens, verá isso em funcionamento, embora o *scaffold* gerado se limite à navegação para frente e para trás na lista de páginas.

A *view* acima também exibe uma complexidade maior, utilizando vários métodos para a geração do seu conteúdo. Notadamente, o método `link_to` que recebe como o texto do *link* e parâmetros adicionais para a geração da URL, que podem ser combinados de várias formas. Por exemplo:

```
<%= link_to :controller => "home" %>
<%= link_to :controller => "contexts", :action => "edit", :id => 2 %>
<%= link_to :controller => "login", :using => "cookies" %>
```

No primeiro caso acima, o método gera a URL raiz para um *controller*, que como vimos é mapeada para a ação `index`. No segundo caso, uma URL completa é retornada. E no terceiro, um parâmetro `using` é adicionado à requisição, gerando a seguinte URL: `/login?using=cookies`.

O método `link_to` possui várias outras capacidades entre as quais gerar confirmações em JavaScript e criar *forms* para submissão confiável de links que modificam destrutivamente seus dados, como pode ser visto nos próprios arquivos geados nesse *scaffold*. Uma olhada na documentação é recomendada para um esclarecimento maior dessas opções.

Um outro conceito interessante apresentado nesse *scaffold* é o de *partials* (parciais), que são fragmentos de páginas que podem ser compartilhados entre *views*, da mesma forma que um *layout* pode usar várias *views*. Veja, por exemplo, a relação entre os dois arquivos abaixo:

Primeiro, a *view* para criação de um novo registro:

```
<h1>New context</h1>

<%= start_form_tag :action => 'create' %>
  <%= render :partial => 'form' %>
  <%= submit_tag "Create" %>
<%= end_form_tag %>

<%= link_to 'Back', :action => 'list' %>
```

Agora, a *view* para edição de um registro:

```
<h1>Editing context</h1>

<%= start_form_tag :action => 'update', :id => @context %>
  <%= render :partial => 'form' %>
  <%= submit_tag 'Edit' %>
<%= end_form_tag %>

<%= link_to 'Show', :action => 'show', :id => @context %> |
<%= link_to 'Back', :action => 'list' %>
```

Note que o código gerado é muito parecido (de fato os dois arquivos e suas ações relacionadas poderiam ser combinados em um único ponto de acesso) e em ambos os arquivos há uma chamada ao método `render`, usando um *partial*. Esse partial se encontra no arquivo `_form.rhtml`, no mesmo diretório. Arquivos que definem *partials* sempre começam com `_` para especificar que são fragmentos. O conteúdo do arquivo é simples:

```
<%= error_messages_for 'context' %>

<!--[form:context]-->
<p><label for="context_name">Name</label><br/>
<%= text_field 'context', 'name' %></p>
<!--[eoform:context]-->
```

Nesse caso, a parte do formulário que é comum tanta à inserção de um contexto quanto à sua edição.

Partials são uma das grandes facilidades do Rails, sendo utilizados principalmente em aplicações Ajax para gerar somente os fragmentos do código necessários para atualizações de partes de uma página. Hoje, com o uso do templates RJS, eles se tornaram ainda mais importantes.

Se você utilizar o *scaffold* gerado, verá que o código é um pouco mais polido, com mensagens de sucesso e paginação, mas com pouco diferença do que poderia ser feito com uma única linha de código, como visto anteriormente. Por isso, a observação anterior de que *scaffolds* são úteis, mas não devem se tornar uma regra na aplicação. Um grande exemplo das limitações é o modo como os *scaffolds* básicos do Rails geram as tabelas de listagens, fazendo invocações diretas aos atributos de um objeto em um *loop* com pouca possibilidade de customização.

Veja o fragmento de código abaixo, proveniente da *view index.rhtml*:

```
<tr>
<% for column in Context.content_columns %>
  <th><%= column.human_name %></th>
<% end %>
</tr>
```

Os cabeçalhos da tabela de listagem são gerados fazendo uma iteração sobre as colunas de conteúdo do modelo de dados em questão, que não incluem colunas avançadas de relacionamento e nem são capazes de fazer a tradução adequada de colunas que contenham valores booleanos, enumerados e tipos de dados próprios do banco. Além disso, o método *human_name* é voltado para o idioma inglês e embora exista a possibilidade de adaptá-lo para outro idioma por meio de *plugins* e outros métodos, as limitações dessas técnicas são logo aparentes e, se cuidado não for tomado, podem levar a problemas de lógica na aplicação, tornando a mesma desnecessariamente complexa e violando os princípios básicos de simplicidade sobre os quais o Rails foi construído.

Aproveitando momentaneamente o código gerado, vamos avançar um pouco em nossa aplicação, fazendo uso de mais uma característica no Rails.

VALIDAÇÕES

Validações, como o próprio nome indica, são um modo de garantir a integridade dos dados em uma aplicação. O modelo de validações que o Rails fornece é bastante completo e pode ser facilmente expandido pelo desenvolvedor caso ele necessite de algo não fornecido por padrão nas bibliotecas.

No caso da nossa classe de dados inicial, precisamos validar pelo menos o fato do usuário ter informado o nome do contexto ao cadastrá-lo.

Para isso, vamos abrir o arquivo da classe, que está em *app/model/context.rb*, e editá-lo, inserindo uma validação simples:

```
class Context < ActiveRecord::Base  
  validates_presence_of :name  
end
```

Se você tentar inserir agora um contexto sem nome, verá o seguinte:

Contexts: create - Firefox

New context

1 error prohibited this context from being saved

There were problems with the following fields:

- Name can't be blank

Name

Create

Back

Done

A menos que o usuário informe o nome do contexto, o Rails não permitirá que o objeto seja salvo. Note que a validação é feita no modelo de dados e refletida na *view* através dos métodos da mesma.

O método `error_messages_for` recolhe todas as mensagens de erro geradas durante uma validação e gera o HTML necessário para exibi-las, que, no caso acima, ainda é formatado pela `stylesheet scaffold.css` criada anteriormente. Da mesma forma, os métodos responsáveis pela geração dos itens de formulário (como `text_field`, `text_area`, `select` e `datetime_select`, entre outros) são capazes de verificar se o campo a que se referem não falhou em alguma validação e encapsular a exibição do campo em uma indicação de erro.

Múltiplas validações podem ser efetuadas em um mesmo campo. Por exemplo, para prevenir a inserção de nomes duplicados, a seguinte condição poderia ser colocada na classe:

```

class Context < ActiveRecord::Base

  validates_presence_of :name
  validates_uniqueness_of :name

end

```

O resultado seria:

New context

1 error prohibited this context from being saved

There were problems with the following fields:

- Name has already been taken

Name
@Home

Create Back Done

As mensagens de erro geradas são padronizadas no próprio Rails, mas podem ser customizadas usando o seguinte formato:

```

class Context < ActiveRecord::Base

  validates_presence_of :name
  validates_uniqueness_of :name, :message => "must be unique"

end

```

Note que o método `error_messages_for` é voltado para o inglês, formando frases que fazem mais sentido nesse idioma, enquanto o nosso português prefere frases mais elaboradas. É fácil substituir o método acima por uma implementação mais interessante do mesmo que atenda ao português e como veremos mais adiante, em outra seção do tutorial.

Um último passo seria atualizar esse *scaffold* para usar o *layout* que criamos. Para isso, remova o arquivo `contexts.rhtml` e atualize o arquivo `application.rhtml` (ambos no diretório `app/views/layouts`) para o seguinte:

```
<html>
<head>
  <title>GTD</title>
  <%= stylesheet_link_tag "default" %>
  <%= stylesheet_link_tag "scaffold" %>
</head>

<body>
  <%= yield %>
</body>

</html>
```

A mudança não é grande, mas significa que nosso *scaffold* agora usa o *layout* da aplicação, recebendo todas as modificações provenientes do mesmo.

Cabe uma nota aqui sobre as ações geradas no *scaffold* e, por extensão, qualquer outra ação a ser criada em uma aplicação.

Um movimento dentro da comunidade do Rails hoje é o uso de métodos REST, que representam uma interpretação de um conjunto de padrões e filosofias de desenvolvimento Web.

Esses métodos usam verbos HTTP mais específicos como PUT e DELETE para efetuar suas ações, ao invés de usar somente os usuais GET e POST. A próxima versão do Rails, ao que tudo indica, virá com um suporte bem forte para esse tipo de uso. O presente tutorial não entrará nesses detalhes sobre isso, mas deixo ao leitor alguns recursos para que ele possa pesquisar mais sobre o assunto:

<http://www.xml.com/pub/a/2005/11/02/rest-on-rails.html>

<http://pezra.barelyenough.org/blog/2006/03/another-rest-controller-for-rails/>

<http://www.agilewebdevelopment.com/plugins/simplyrestful>

Agora que vimos com o básico funciona no Rails, podemos partir para tentar a nossa própria implementação, aprendendo mais sobre com as coisas funcionam.

UM SEGUNDO CONTROLLER

Vamos criar o cadastro de projetos—que também é bem simples—de forma mais manual para entendermos como uma aplicação comum processa seus dados. Para facilitar, vamos combinar a criação e edição de um registro em uma única ação.

O primeiro passo é atualizar o modelo de dados para efetuar validações. No caso no nosso modelo de dados

para projetos (que está no arquivo `app/models/project.rb`), precisamos de validações igualmente simples. A classe ficaria, assumindo que o atributo `description` é opcional, assim:

```
class Project < ActiveRecord::Base  
  validates_presence_of :name  
  validates_uniqueness_of :name  
end
```

Um segundo passo é gerar um *controller* para lidar com as requisições relacionadas ao cadastro de projetos. Dessa vez, vamos usar o comando de geração do *controllers* especificando já as ações que desejamos criar automaticamente.

Basicamente, queremos as ações *index*, *list*, *edit* e *delete*. A ação *index* existe para não precisarmos de uma regra de roteamento explícita mas será meramente uma invocação da ação *list*. E ação *edit* resumirá todas as ações de criação e inserção, simplificando o código.

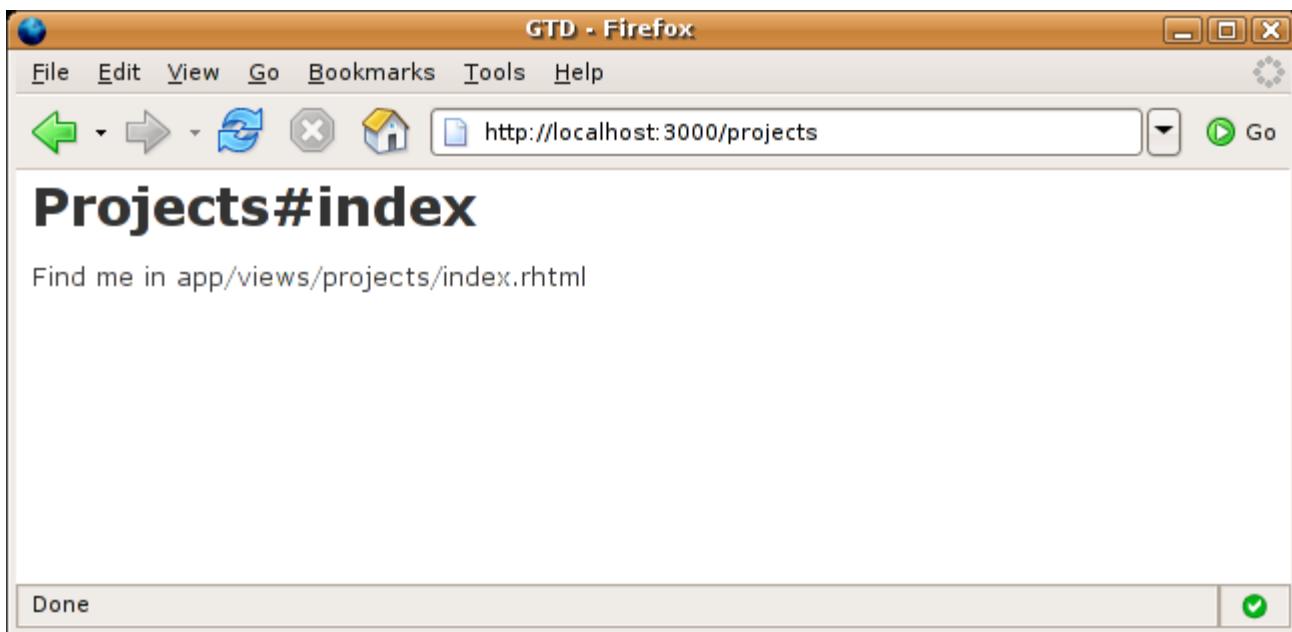
O comando para gerar *controller* é o seguinte:

```
ronaldo@minerva:~/tmp/gtd$ script/generate controller projects index list edit delete  
exists app/controllers/  
exists app/helpers/  
create app/views/projects  
exists test/functional/  
create app/controllers/projects_controller.rb  
create test/functional/projects_controller_test.rb  
create app/helpers/projects_helper.rb  
create app/views/projects/index.rhtml  
create app/views/projects/list.rhtml  
create app/views/projects/edit.rhtml  
create app/views/projects/delete.rhtml
```

Como você pode ver, os arquivos das *views* foram automaticamente criados, e se você abrir o arquivo do *controller* em si (`app/controllers/projects_controller.rb`) você verá o esqueleto que foi gerado:

```
class ProjectsController < ApplicationController  
  def index  
  end  
  
  def list  
  end  
  
  def edit  
  end  
  
  def delete  
  end  
end
```

Invocando o *controller* em seu navegador, você verá o seguinte:



Nós agora temos um *controller* que responde perfeitamente às nossas ações, com *views* já associadas, prontas para o uso.

Como a ação *index* é somente uma invocação da ação *list*, podemos remover o arquivo da *view* da mesma que está em `app/views/projects/index.rhtml` e começar a editar as ações. O método *index* é muito simples:

```
class ProjectsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
  end

  def edit
  end

  def delete
  end
end
```

O que essas duas linhas nos dizem são: primeiro, invoque o método *list*, e depois renderize a *view* da ação *list*. Mesmo representando uma ação em si, *list* não passa de um método comum da classe. É somente a mágica do Rails que o torna uma ação, e, sendo assim, ele pode ser invocado normalmente, como qualquer outra método. A segunda linha é necessária porque o mecanismo automático de identificação de *views* procuraria a *view* relacionada a *index* e não a *list*. Precisamos, então, explicitar a que será usada pela

aplicação.

Antes de construirmos a ação *list*, precisamos providenciar um modo de inserirmos registros do banco. Para isso, precisamos criar a nossa ação *edit*. Temos duas situações possíveis: ou o usuário está criando um novo registro, ou ele está atualizando um registro existente. Obviamente, a única diferença entre esses dois métodos é a existência ou não do objeto. E a primeira coisa a ser feita tanto na criação quanto na edição de um objeto é exibir o formulário do mesmo para preenchimento.

Como ações relacionadas somente a exibição geralmente são associadas ao método HTTP GET enquanto ações que modificam dados e dependem de formulários são invocadas a partir do método POST, vamos usar essa diferenciação para identificar se estamos salvando o objeto ou somente exibindo o formulário para edição. E mais, vamos usar a existência ou não do parâmetro *id* para identificar se estamos lidando com um novo objeto ou com um objeto já existente. A primeira versão do nosso método ficaria assim, então:

```
class ProjectsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
  end

  def edit
    if params[:id]
      @project = Project.find(params[:id])
    else
      @project = Project.new
    end
    if request.post?
      end
    end
  end

  def delete
  end
end
```

O primeiro teste do método verifica se um parâmetro *id* foi passado. No Ruby, é possível testar a existência como na linguagem C. Se o objeto for nulo—ou `nil` em Ruby—o teste retornará falso. Caso contrário, retornará verdadeiro. Se você quiser jogar pelo lado da legibilidade, a linha poderia ser convertida para `if params[:id].nil?`, com uma inversão das condições. Ou você pode usar `unless params[:id].nil?` sem inverter as condições, já que essa declaração testa o contrário de uma declaração `if`.

A versão do teste que usamos acima é um pouquinho mais eficiente, mas não o suficiente para fazer qualquer diferença geral da aplicação e você pode preferir a legibilidade maior.

No código acima, dependendo da existência ou não do parâmetro, tentamos carregar o objeto do banco ou criar um novo objeto, preparando o caminho para a nossa *view*.

O segundo teste do método verifica se estamos recebendo uma requisição POST. Como mencionamos anteriormente, é isso que vamos usar para saber se estamos simplesmente exibindo o formulário (seja vazio ou não), ou se está na hora de persistir o objeto para o banco. Mais à frente colocaremos esse código.

Caso você ache que o nome *edit* para esse método é meio enganoso quando um novo registro está sendo criado, você pode usar regras de roteamento genéricas para especificar que invocações a ações chamadas *create*, *update*, e *save* são redirecionadas para a mesma ação *edit*. Separar ou não as suas ações é uma decisão que depende obviamente do que você está fazendo no momento. A tendência com o uso de REST, inclusive, é de ações bem específicas, que respondem a somente um tipo de verbo HTTP.

Agora que o nosso método começou a tomar forma, precisamos editar a `view edit.html` para exibir o nosso formulário.

O arquivo presente está assim:

```
<h1>Projects#edit</h1>
<p>Find me in app/views/projects/edit.rhtml</p>
```

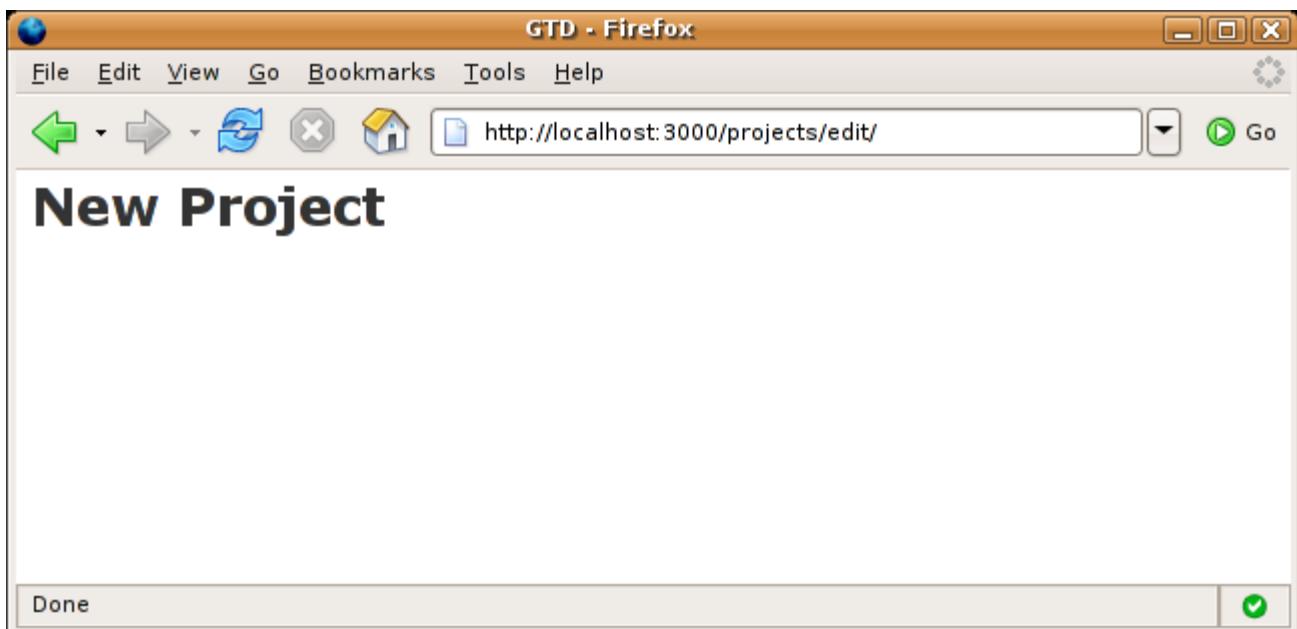
Precisamos indicar ao usuário se estamos editando um registro existente ou adicionando um novo registro no próprio cabeçalho da página. Para isso podemos usar o seguinte código:

```
<h1><% if @project.new_record? %>New<% else %>Editing<% end %> Project</h1>
```

Note que estamos usando a variável `@project` que criamos no método `edit`. O `@` indica uma variável de instância, que o Rails automaticamente propaga para qualquer `view` sendo processada a partir da ação em que a variável foi definida.

Na linha acima, estamos usando o método `new_record?`, presente em todas as classes de dados derivadas do *ActiveRecord* para identificar se estamos lidando com um novo registro ou com um registro já existente. Esse método retorna verdadeiro enquanto a classe não for salva pela primeira vez. Com base nisso, exibimos o nosso cabeçalho.

Se você rodar a página, invocando diretamente a URL `/projects/edit`, verá que já temos algo funcionando:



Obviamente, se você tentar usar um *id* qualquer você verá um erro a menos que tenha inserido aquele registro no banco de dados.

O próximo passo agora é criar o formulário de dados. Para isso, vamos editar a nossa *view*:

```
<h1><% if @project.new_record? %>New<% else %>Editing<% end %> Project</h1>
<%= start_form_tag :action => "edit", :id => @project %>
<%= end_form_tag %>
```

O método `start_form_tag` funciona de maneira muito similar o método `link_to`, explicado anteriormente. No caso acima, estamos gerando um formulário apontando para a ação `edit` e passando o *id* do objeto que estamos usando, seja novo ou não.

Duas coisas são interessantes nesse chamada: primeiro, o Rails é inteligente o bastante para perceber que você está passando o objeto como se fosse o seu *id* e gerar o código necessário automaticamente; segundo, caso o objeto não tenha sido salvo, o Rails suprimirá automaticamente o *id* sem que você precise fazer qualquer teste.

O outro método chamado, `end_form_tag`, simplesmente gera o fechamento do elemento `form` na página.

Com o cabeçalho do nosso formulário gerado, podemos agora criar campos para edição. Um projeto possui um nome e um descrição. O primeiro atributo é um texto simples, de uma linha somente, enquanto o segundo é um texto que pode ter múltiplas linhas. Editando o nosso arquivo, teremos o seguinte:

```

<h1><% if @project.new_record? %>New<% else %>Editing<% end %> Project</h1>

<%= start_form_tag :action => "edit", :id => @project %>

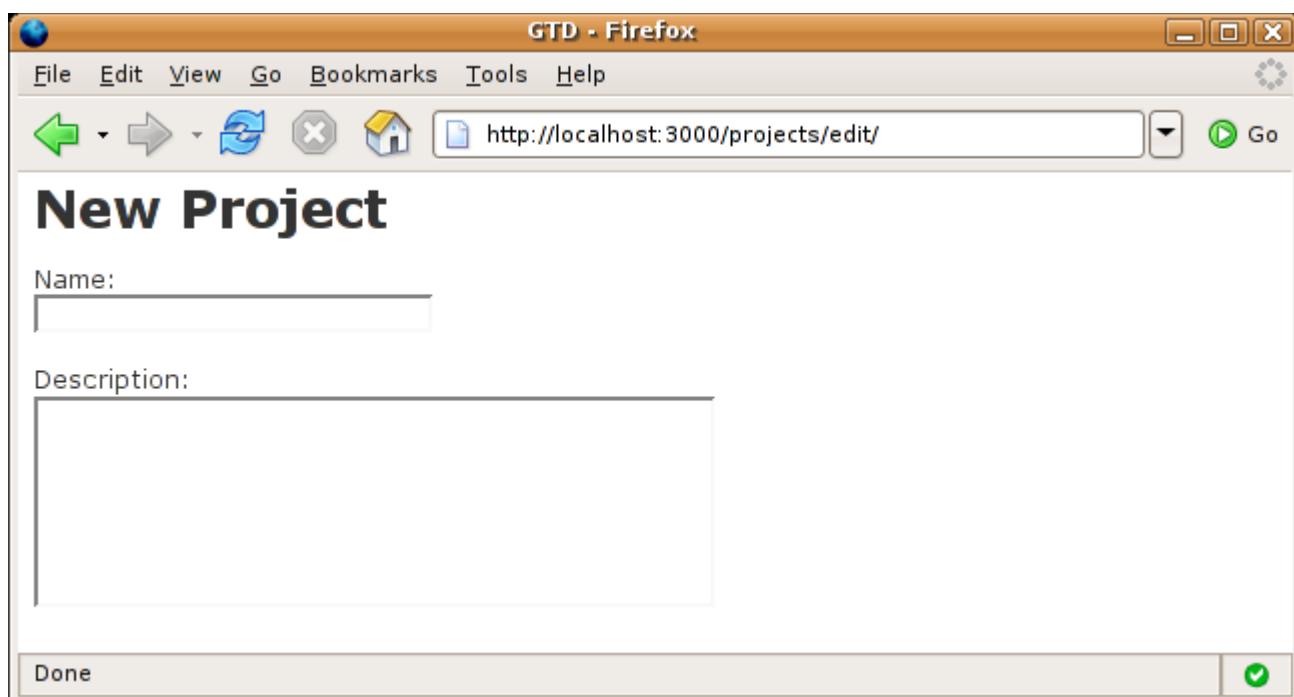
<p>
  <label for="project_name">Name:</label><br>
  <%= text_field "project", "name" %>
</p>

<p>
  <label for="project_description">Description:</label><br>
  <%= text_area "project", "description", :rows => 5 %>
</p>

<%= end_form_tag %>

```

Essa modificação nos dá o seguinte:



Os métodos `text_field` e `text_area` são responsáveis pela geração dos elementos de formulário vistos acima. O Rails possui dezenas desses métodos, capazes de gerar várias combinações possíveis para os tipos de dados suportados automaticamente e permitir extensões se você precisar de algo mais customizado. E você sempre pode gerar o seu código manualmente em situações especiais.

Veja que cada método recebe um objeto e um atributo a ser gerado. No caso desses métodos, você não precisa usar o `@` diretamente: basta passar o nome do objeto e o método saberá recuperá-lo do *controller* que está sendo executado.

Se você olhar o HTML gerado, verá o seguinte:

```

<p>
  <label for="project_name">Name:</label><br>
  <input id="project_name" name="project[name]" size="30" type="text" />
</p>

<p>
  <label for="project_description">Description:</label><br>
  <textarea cols="40" id="project_description" name="project[description]" rows="5"></textarea>
</p>

```

Veja que os métodos geram um formato similar de elementos, criando atributos *name* e *id* específicos para facilitar a vida do desenvolvedor. O atributo *id* pode ser associado a um elemento *label* como demonstrado e o atributo *name* é o que é enviado ao Rails, gerando automaticamente uma tabela *hash* dos dados submetidos pelo formulário que a ação pode usar.

Dentro de um *controller* você será capaz de usar `params[:project]` para acessar diretamente essa tabela *hash*, como você verá adiante.

O próximo passo, agora, é adicionar ações para salvar ou cancelar a edição. Poderíamos ter algo assim:

```

<h1><% if @project.new_record? %>New<% else %>Editing<% end %> Project</h1>

<%= start_form_tag :action => "edit", :id => @project %>

<p>
  <label for="project_name">Name:</label><br>
  <%= text_field "project", "name" %>
</p>

<p>
  <label for="project_description">Description:</label><br>
  <%= text_area "project", "description", :rows => 5 %>
</p>

<p>
  <%= submit_tag "Save" %> or <%= link_to "Cancel", :action => "list" %>
</p>

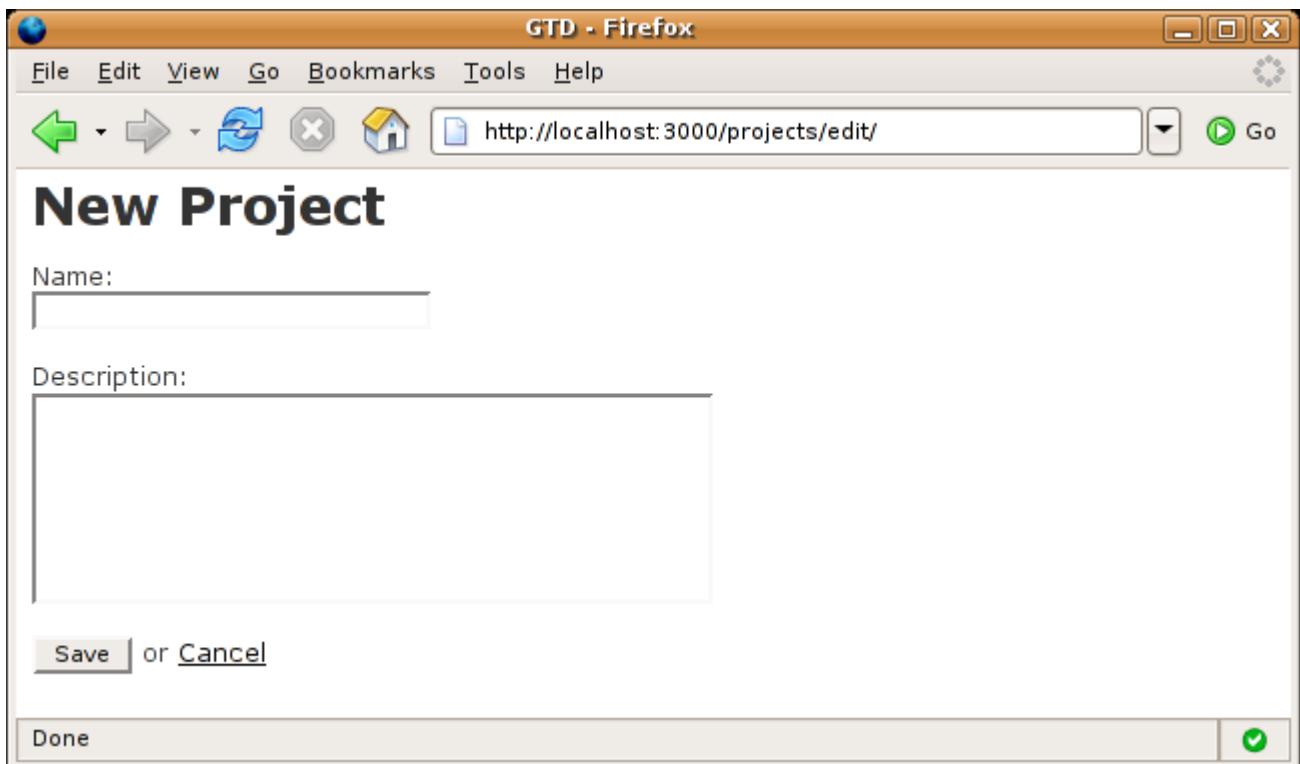
<%= end_form_tag %>

```

O formato acima é uma preferência de muitas aplicações Rails e reflete o fato de que a ação para salvar deveria ser submetida pelo formulário enquanto a ação de cancelamento seria simplesmente um retorno a um estado anterior. Mais do que isso, o método acima evita que tenhamos que descobrir qual botão foi pressionado em nosso formulário. Embora o código para isso seja bem simples, repetí-lo em cada formulário se tornaria rapidamente tedioso, além de forçar uma mistura de apresentação com lógica que queremos evitar a todo custo.

Esse é um padrão que você verá repetidamente em aplicações atuais e que faz bastante sentido do ponto de vista de usabilidade.

O resultado seria:



Temos agora um formulário completo funcionando. Precisamos simplesmente salvar o que será submetido. Voltando ao nosso *controller*, ficamos com o seguinte:

```
class ProjectsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
  end

  def edit
    if params[:id]
      @project = Project.find(params[:id])
    else
      @project = Project.new
    end
    if request.post?
      @project.attributes = params[:project]
      if @project.save
        redirect_to :action => "list"
      end
    end
  end

  def delete
  end
end
```

Todo objeto que representa uma classe de dados no Rails possui um atributo chamado `attributes`, que representa uma coleção das colunas da tabela equivalentes. Esse atributo aceita receber uma tabela *hash* para preenchimento de seus valores. Como mencionado anteriormente, é aqui que os dados gerados pelo Rails na submissão do formulário vem a calhar, servindo para associação direta no objeto. Cada item existente em `params[:project]` no código acima, como, por exemplo, `params[:project][:name]`, preencherá seu item correspondente sem necessidade de código adicional.

Continuando o código, tentamos salvar o objeto. Se isso tem sucesso (ou seja, se nenhuma validação falha) redirecionamos para a página de listagem. Caso contrário, exibimos o formulário novamente. Nesse caso, note que o Rails usará a mesma *view*, seguindo o comportamento padrão, efetivamente mostrando o formulário preenchido com os dados postados. Isso acontece porque os métodos `text_field` e `text_area` (e os demais) preenchem automaticamente os elementos de formulário com os valores existentes no objeto que receberam. Isso torna a nossa tarefa bem simples.

Se você tentar salvar um formulário sem preencher nada, verá o seguinte agora:

The screenshot shows a Firefox browser window titled "GTD - Firefox". The address bar displays the URL "http://localhost:3000/projects/edit/". The main content area is a form titled "New Project". It has two input fields: "Name:" and "Description:". The "Name:" field is empty and has a red border around it, indicating it is required. Below the form are two buttons: "Save" and "Cancel", followed by a "Done" button with a checked checkbox.

Notamos que a validação está funcionando, mas nenhum erro é exibido. Para isso, precisamos da seguinte modificação em nossa *view*:

```

<h1><% if @project.new_record? %>New<% else %>Editing<% end %> Project</h1>

<%= error_messages_for "project" %>

<%= start_form_tag :action => "edit", :id => @project %>

<p>
  <label for="project_name">Name:</label><br>
  <%= text_field "project", "name" %>
</p>

<p>
  <label for="project_description">Description:</label><br>
  <%= text_area "project", "description", :rows => 5 %>
</p>

<p>
  <%= submit_tag "Save" %> or <%= link_to "Cancel", :action => "list" %>
</p>

<%= end_form_tag %>

```

O que nos dá:

New Project

1 error prohibited this project from being saved

There were problems with the following fields:

- Name can't be blank

Name:

Description:

Save or Cancel

Done

Se você salvar um objeto agora, verá que somos redirecionados para a ação *list* que ainda precisamos criar.

Uma listagem é algo bem simples e queremos exibir, inicialmente, somente o nome do projeto. Para isto, vamos criar algo bem básico:

```
<h1>Projects</h1>

<table border="1" cellpadding="5" cellspacing="1">
  <tr>
    <th>Name</th>
  </tr>
</table>
```

Isso nos dá um tabela simples. Agora, queremos exibir os registros criados. Precisamos, obviamente, buscar esses objetos no banco. Modificando o *controller* ficamos com algo assim:

```
class ProjectsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @projects = Project.find :all, :order => "name"
  end

  def edit
    if params[:id]
      @project = Project.find(params[:id])
    else
      @project = Project.new
    end
    if request.post?
      @project.attributes = params[:project]
      if @project.save
        redirect_to :action => "list"
      end
    end
  end

  def delete
  end
end
```

Um variável chamada `@projects` receberá uma lista de todos projetos ordenados pelo atributo *name*. O método `find`, já tratado anteriormente, recebe como primeiro parâmetro uma especificação do que retornar ou um *id* específico. No caso acima, queremos todos registros (*all*). O segundo parâmetro, nomeado, recebe a especificação de ordenação.

Um detalhe da chamada acima. Muitas vezes, em código Ruby, você verá um seqüência de parâmetros nomeados no fim de uma chamada. O Ruby não possui parâmetros nomeados em si, mas é capaz de simular isso com o uso de uma tabela *hash* com último parâmetro de um método. Quando um método usa essa técnica, o Ruby automaticamente permite que os parâmetros sejam declarados de forma livre, pelo nome, e

os combina no momento da chamada em uma tabela *hash* que é então passada como parâmetro. Assim, o primeiro parâmetro do método *find* acima é um símbolo e o segundo parâmetro é uma tabela *hash* cujo único elemento é um par definido por um símbolo e uma *string*.

Agora, já podemos usar esses objetos em uma *view*:

```
<h1>Projects</h1>



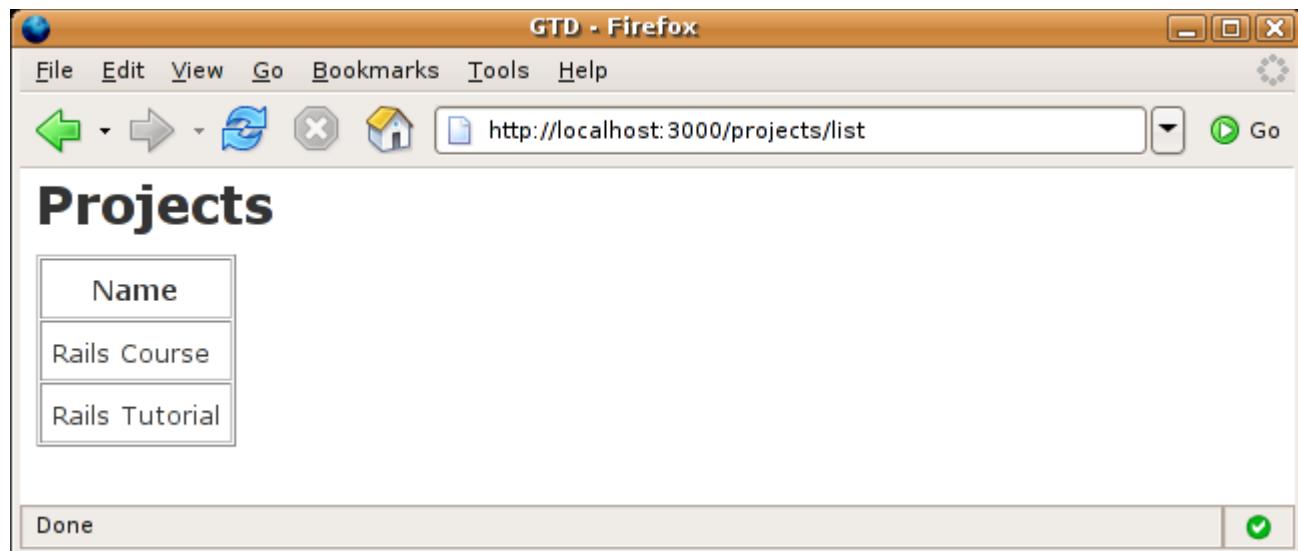



```

No caso acima, o método *find* retorna um *array* que pode ou não conter itens.

Uma coisa a manter em mente aqui é que, para caso de uso desse método, o retorno pode ser diferente. Se você estiver usando *find* com *:first*, o retorno será um objeto ou o valor *nil*. Se você estiver passando um *id* específico, um objeto será retornado ou um erro será gerado. Lembre-se disso sempre que usar o método em suas variações. As variações servem para cobrir as situações mais comuns, e você sempre pode escrever seus próprios métodos usando as combinações acima para obter o resultado que deseja.

Com dois projetos inseridos, o resultado é:



Podemos agora modificar a nossa *view* para permitir algumas ações sobre esses objetos. Por exemplo:

```

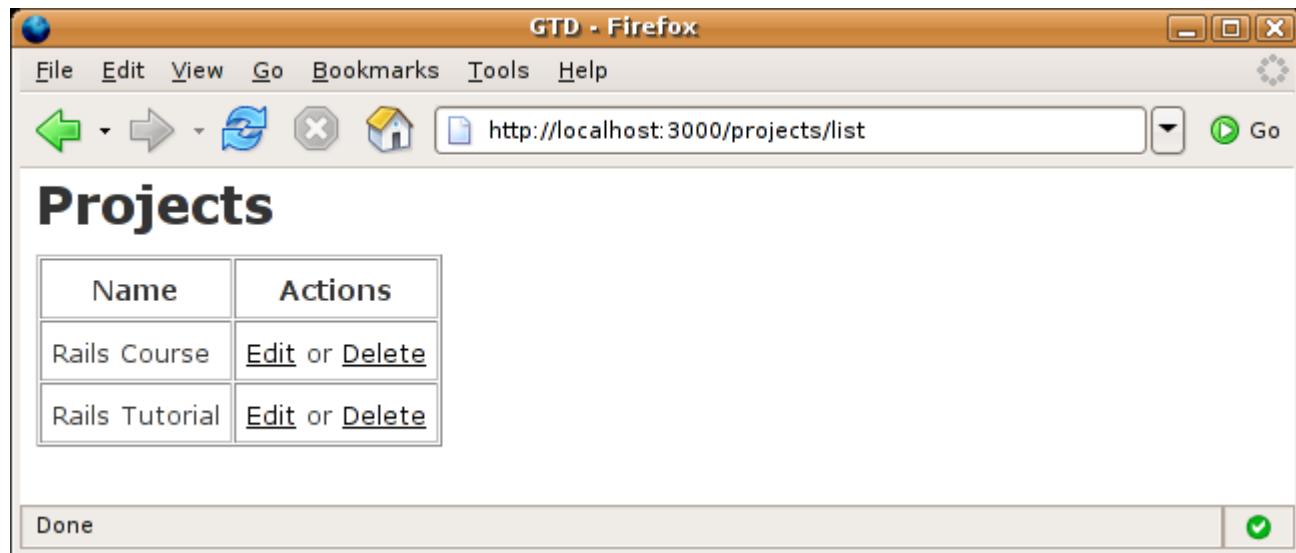
<h1>Projects</h1>

<table border="1" cellpadding="5" cellspacing="1">
  <tr>
    <th>Name</th>
    <th>Actions</th>
  </tr>
  <% @projects.each do |project| %>
  <tr>
    <td><%= project.name %></td>
    <td>
      <%= link_to "Edit", :action => "edit", :id => project %> or
      <%= link_to "Delete", :action => "delete", :id => project %>
    </td>
  </tr>
  <% end %>
</table>

```

Como você deve ter notado pelas chamadas a `link_to` na *view* acima e em outros pontos do código já exibido, o Rails é capaz de derivar os elementos da uma URL dos parâmetros que estão sendo usados no momento.

O resultado da aplicação da *view* acima é visto abaixo:

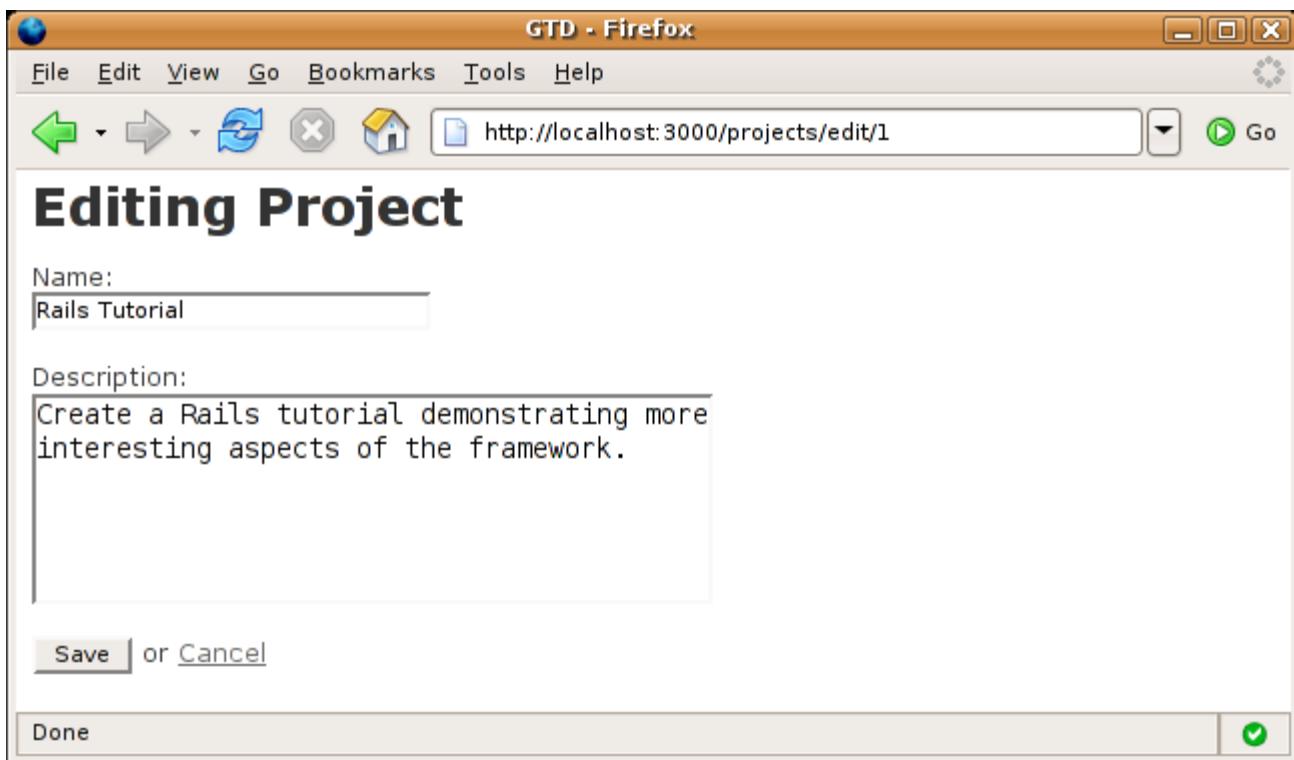


The screenshot shows a Firefox browser window titled "GTD - Firefox". The address bar displays the URL "http://localhost:3000/projects/list". The main content area shows a table with the following data:

Name	Actions
Rails Course	Edit or Delete
Rails Tutorial	Edit or Delete

Below the table, there is a "Done" button with a checked checkbox next to it.

Clicando na edição de um dos itens acima, temos:



Como é possível ver, o cabeçalho da página mudou para indicar uma edição e os campos já vieram preenchidos. Você já pode alterar cada objeto sem problemas. Precisamos agora de refinar alguns aspectos de nosso *controller*.

Precisamos de um *link* para criar um novo registro. Isso é facilmente conseguido:

```
<h1>Projects</h1>

<p><%= link_to "New Project", :action => "edit" %></p>

<table border="1" cellpadding="5" cellspacing="1">
  <tr>
    <th>Name</th>
    <th>Actions</th>
  </tr>
  <% @projects.each do |project| %>
  <tr>
    <td><%= project.name %></td>
    <td>
      <%= link_to "Edit", :action => "edit", :id => project %> or
      <%= link_to "Delete", :action => "delete", :id => project %>
    </td>
  </tr>
  <% end %>
</table>
```

Um dos problemas em deixar que um registro seja removido com um simples invocação é que o usuário não tem como reverter a ação. Podemos adicionar um mínimo de proteção com a seguinte alteração:

```

<h1>Projects</h1>

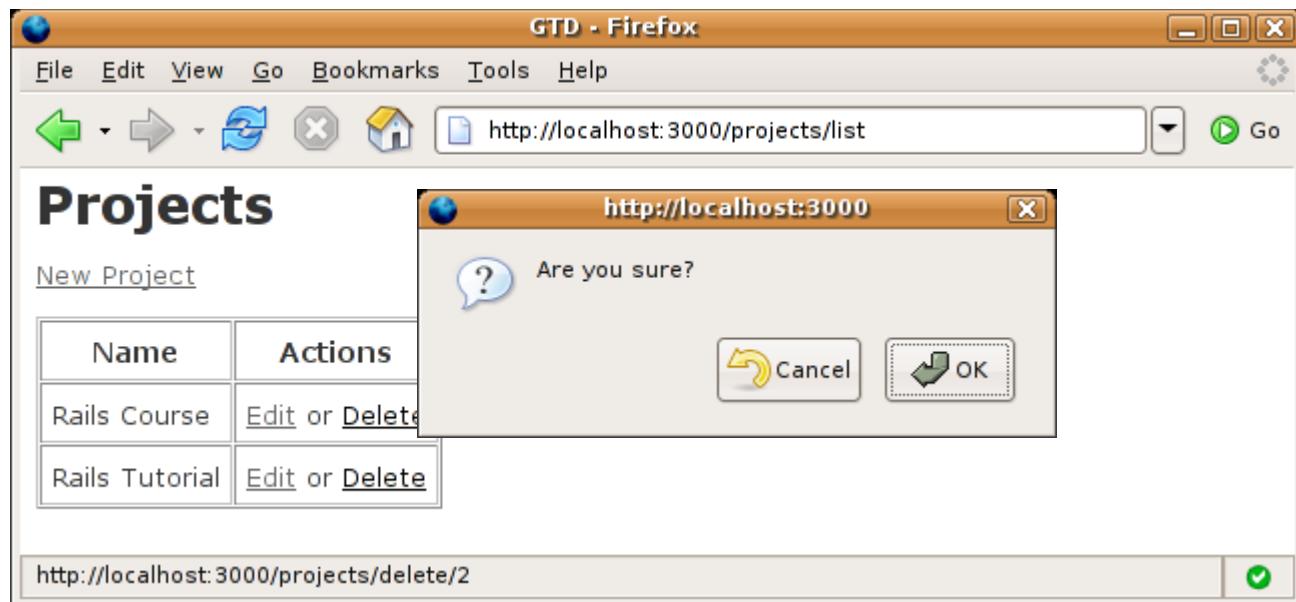
<p><%= link_to "New Project", :action => "edit" %></p>

<table border="1" cellpadding="5" cellspacing="1">
  <tr>
    <th>Name</th>
    <th>Actions</th>
  </tr>
  <% @projects.each do |project| %>
  <tr>
    <td><%= project.name %></td>
    <td>
      <%= link_to "Edit", :action => "edit", :id => project %> or
      <%= link_to "Delete", { :action => "delete", :id => project }, :confirm => "Are you sure?" %>
    </td>
  </tr>
  <% end %>
</table>

```

Note que, por estamos usando um terceiro parâmetro do método `link_to`, precisamos de adicionar chaves ao redor do segundo para que o Ruby não exerça o seu comportamento padrão e concatene todos os parâmetros livres em um só. No caso acima, tanto o segundo parâmetro da chamada como o terceiro são tabelas *hash*, embora somente a segunda esteja explicitamente indicada.

O resultado das modificações acima é o seguinte, como podemos ver ao clicar na ação para remover o projeto:



A ação `delete` também pode ser protegida de acesso GET diretos com o uso de outra característica do Rails que é a transformação de *links* em formulários caso a ação seja bem simples. O resultado visual não é muito interessante e você pode observá-lo no código do *scaffold* que foi gerado anteriormente para o modelo de dados de contextos.

Vamos agora inserir a ação para excluir registros, que é bem simples:

```
class ProjectsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @projects = Project.find :all, :order => "name"
  end

  def edit
    if params[:id]
      @project = Project.find(params[:id])
    else
      @project = Project.new
    end
    if request.post?
      @project.attributes = params[:project]
      if @project.save
        redirect_to :action => "list"
      end
    end
  end

  def delete
    Project.find(params[:id]).destroy
    redirect_to :action => "list"
  end
end
```

Como essa ação não possui nenhuma *view*, o arquivo gerado `delete.rhtml` também pode ser excluído.

Uma outra alteração que pode ser feita é a exibição de mensagens de dados para algumas ações. Para isso, vamos usar uma outra característica dos Rails, as variáveis *flash*. Essas variáveis são similares a variáveis de sessão, mas somente persistem de uma página para outra. Uma vez que a requisição para a qual foram propagadas acaba, elas são automaticamente removidas do contexto de execução.

Vamos modificar nosso *controller* mais uma vez:

```
class ProjectsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @projects = Project.find :all, :order => "name"
  end

  def edit
    if params[:id]
      @project = Project.find(params[:id])
    else
      @project = Project.new
    end
    if request.post?
```

```

@project.attributes = params[:project]
if @project.save
  flash[:notice] = "The project was successfully saved"
  redirect_to :action => "list"
end
end
end

def delete
  Project.find(params[:id]).destroy
  flash[:notice] = "The project was successfully deleted"
  redirect_to :action => "list"
end
end

```

Podemos agora usar essas variáveis em nossa *view* para a ação *list*, que é para onde as duas ações em questão retornam:

```

<h1>Projects</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New Project", :action => "edit" %></p>

<table border="1" cellpadding="5" cellspacing="1">
  <tr>
    <th>Name</th>
    <th>Actions</th>
  </tr>
  <% @projects.each do |project| %>
  <tr>
    <td><%= project.name %></td>
    <td>
      <%= link_to "Edit", :action => "edit", :id => project %> or
      <%= link_to "Delete", { :action => "delete", :id => project }, :confirm => "Are you sure?" %>
    </td>
  </tr>
  <% end %>
</table>

```

Uma pequena observação que cabe aqui é o uso de símbolos ao invés de *strings* na referência a tabelas *hash*. Símbolos são sempre uma forma de acesso mais interessante por apontarem sempre para uma única instância e poderem ser otimizados pelo interpretador. O Rails geralmente permite as duas formas de acesso para suas variáveis internas representadas com tabelas *hash*, mas é sempre bom escolher um dos métodos de acesso e manter um padrão em relação ao mesmo. Isso facilita a legibilidade da aplicação e evita o aparecimento de erros proveniente da mistura dos dois tipos de acesso.

O resultado é o seguinte, após uma ação de edição:

The project was successfully saved

[New Project](#)

Name	Actions
Rails Course	Edit or Delete
Rails Tutorial	Edit or Delete

Done

A variável `flash[:notice]` está disponível somente naquela requisição como você pode verificar recarregando a página.

EXTENDENDO UM MODELO DE DADOS

Algo que podemos fazer agora, para aproveitar o nosso *controller* é extender o nosso modelo de dados. Vamos dizer que precisamos saber se um projeto está ativo ou não. O nosso modelo de dados não contém esse atributo, mas podemos gerar uma migração para isso.

```
ronaldo@minerva:~/tmp/gtd$ script/generate migration add_status_to_project
exists db/migrate
create db/migrate/004_add_status_to_project.rb
```

Essa é uma migração um pouco diferente porque adiciona uma coluna a um modelo de dados. Editando o arquivo da migração, teríamos algo assim:

```
class AddStatusToProject < ActiveRecord::Migration
  def self.up
    add_column :projects, :active, :boolean
    Project.reset_column_information
    Project.update_all "active = 1"
  end

  def self.down
    remove_column :projects, :active
  end
end
```

Nesse caso, estamos adicionando uma coluna e automaticamente atualizando para um valor que achamos

mais indicado.

Como o nosso modelo está sendo modificado, precisamos usar o método `reset_column_information` para que o Rails releia as tabelas e recarregue os modelos imediatamente. Na verdade, isso não é estritamente necessário pelo uso do método `update_all`, mas é interessante saber que esse uso pode ser necessário em alguns casos. O método `update_all` atualiza todos os registros usando o fragmento de SQL passado como parâmetro., pulando quaisquer validações existentes—portanto, use-o com cuidado.

Após rodar o comando `rake db:migrate`, veremos o nosso banco atualizado completamente. Note que, nesse caso, a migração para um versão anterior simplesmente remove a coluna.

USANDO HELPERS

Agora precisamos atualizar o nosso *controller* e nossas *views*. O primeiro passo é atualizar a listagem para exibir a situação do projeto. Para isso vamos utilizar um recurso do Rails chamado de *helpers*.

Helpers são classes associadas a cada *controller* que contém funções utilitárias que são automaticamente exportadas para as *views* associadas ao mesmo. Cada *controller* tem o seu próprio *helper* e, a exemplo dos *controllers*, todos os *helpers* herdam métodos de uma classe base, que está definida no arquivo `app/helpers/application_helper.rb`. Olhando no diretório desse arquivo você pode ver que para cada *controller*, um *helper* já foi gerado.

Precisamos de um método que receba um valor booleano e retorne um representação humana disso. Como talvez precisemos usar esse método em outras *views* que não as da classe de projetos, seria mais interessante inserir o método no *helper* global da aplicação, modificando o arquivo `application_helper.rb` presente no diretório `app/helpers`. Teríamos, então, algo assim:

```
module ApplicationHelper
  def yes_or_no?(value)
    value ? "Yes" : "No"
  end
end
```

O método `yes_or_no?` está agora disponível para qualquer *view* na aplicação. Vamos usá-lo, modificando a nossa *view*:

```
<h1>Projects</h1>
<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>
<p><%= link_to "New Project", :action => "edit" %></p>
```

```


| Name                | Active                             | Actions                                                                                                                                                      |
|---------------------|------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <%= project.name %> | <%= yes_or_no?(project.active?) %> | <%= link_to "Edit", :action => "edit", :id => project %> or<br><%= link_to "Delete", { :action => "delete", :id => project }, :confirm => "Are you sure?" %> |


```

Note que podemos agora chamar o método definido no *helper* e também que temos um novo atributo na classe.

Uma convenção especial do Rails é que, se um atributo representar um valor booleano, você pode acrescentar um ponto de interrogação após o método para que ele retorne verdadeiro ou falso diretamente. Isso acontece porque nem todos bancos de dados possuem campos booleanos nativos por padrão. No caso do MySQL, por exemplo, valores booleanos são representados por campos inteiros contendo zero ou um. O uso da interrogação facilita a visualização e uso do campo.

Em várias situações, o Rails é capaz de detectar automaticamente o valor booleano, e o ponto de interrogação não é estritamente necessário. Mas é uma boa convenção fazer isso. Não só o código fica mais legível de imediato, como qualquer desenvolvedor Rails será capaz de dizer o tipo de dados armazenado no campo.

O resultado agora é:

Name	Active	Actions
Rails Course	Yes	Edit or Delete
Rails Tutorial	Yes	Edit or Delete

Obviamente, precisamos editar o fato do projeto estar ativo ou não. Vamos alterar nossa view de edição:

```
<h1><% if @project.new_record? %>New<% else %>Editing<% end %> Project</h1>
<%= error_messages_for "project" %>
<%= start_form_tag :action => "edit", :id => @project %>
<p>
  <label for="project_name">Name:</label><br>
  <%= text_field "project", "name" %>
</p>
<p>
  <label for="project_description">Description:</label><br>
  <%= text_area "project", "description", :rows => 5 %>
</p>
<p>
  <label for="project_active">Active:</label><br>
  <%= select "project", "active", [["Yes", true], ["No", false]] %>
</p>
<p>
  <%= submit_tag "Save" %> or <%= link_to "Cancel", :action => "list" %>
</p>
<%= end_form_tag %>
```

Essa é uma das maneiras de editar o atributo, usando um elemento *select*.

O método `select` gera automaticamente o código HTML necessário, recebendo parâmetros similares aos outros métodos e uma lista de valores a serem usados como escolhas. Esses valores são informados como um *array* de *arrays* onde cada item representa um par, contendo a descrição e o valor a ser atribuído. No caso acima, temos descrições *Yes* e *No*, correspondendo a valores *true* e *false*.

O Rails define uma série de outros métodos capaz de gerar elementos *select*. Alguns desses métodos são especializados em gerar coleções aninhadas, por exemplo, enquanto outros geram diversos formatos de data. Antes de experimentar o método genérico acima, consulte a documentação para ver se existe algum método que lhe atenda.

Lembre-se que, para valores booleanos, para que a atribuição funcione automaticamente, os valores *true* e *false* devem ser necessariamente usados. Caso contrário, a conversão automática de tipos do Ruby entra em ação e os dados não são processados apropriadamente.

O resultado pode ser visto abaixo e testando você verá que ele funciona perfeitamente, alterando os valores de acordo com sua escolha sem necessidade de qualquer codificação adicional:

The screenshot shows a Firefox browser window titled "GTD - Firefox". The address bar displays the URL "http://localhost:3000/projects/edit/2". The main content area contains a form titled "Editing Project". The form fields are as follows:

- Name:
- Description:
- Active:
- Buttons: or [Cancel](#)
- Status:

Existem pelo menos duas outras diretas maneiras de fazer a edição de um valor booleano. Uma seria usando *radio buttons* e *checkboxes*. Testar essas maneiras fica como um exercício para o leitor, lembrando que o princípio é o mesmo.

Uma última coisa seria modificar o *controller* para gerar paginação automática. Isso é facilmente conseguido com duas atualizações.

Uma no controller:

```
class ProjectsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @project_pages, @projects = paginate :projects, :per_page => 5, :order => "name"
  end

  def edit
    if params[:id]
      @project = Project.find(params[:id])
    else
      @project = Project.new
    end
    if request.post?
      @project.attributes = params[:project]
      if @project.save
        flash[:notice] = "The project was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def delete
    Project.find(params[:id]).destroy
    flash[:notice] = "The project was successfully deleted"
    redirect_to :action => "list"
  end
end
```

E outra na view:

```
<h1>Projects</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New Project", :action => "edit" %></p>

<table border="1" cellpadding="5" cellspacing="1">
  <tr>
    <th>Name</th>
    <th>Active</th>
    <th>Actions</th>
  </tr>
  <% @projects.each do |project| %>
  <tr>
    <td><%= project.name %></td>
    <td><%= yes_or_no?(project.active?) %></td>
    <td>
      <%= link_to "Edit", :action => "edit", :id => project %> or
      <%= link_to "Delete", { :action => "delete", :id => project }, :confirm => "Are you sure?" %>
    </td>
  </tr>
  <% end %>
</table>

<p><%= pagination_links(@project_pages) %></p>
```

A modificação no *controller* usa um método que já mencionamos anteriormente, `paginate`, para encapsular o processo de paginar os registros e gerar uma listagem de páginas (aqui separadas em cinco registros por página). A modificação na *view*, por sua vez, simplesmente cria uma lista de páginas com base na informação retornada pelo método de paginação.

Os métodos usados são razoavelmente simples, mas, infelizmente, estão entre dois dos mais ineficientes do Rails e devem ser usados com cuidado. No momento, entretanto, com a quantidade de dados que temos, eles não chegam a representar um problema. O resultado depois pode ser visto abaixo.

Visualizando a primeira página, temos:

The screenshot shows a Firefox browser window titled "GTD - Firefox". The address bar displays the URL "http://localhost:3000/projects/list". The main content area shows a heading "Projects" and a sub-heading "New Project". Below this is a table listing five projects:

Name	Active	Actions
Build a House	Yes	Edit or Delete
Compose a Symphony	Yes	Edit or Delete
Plant a Tree	Yes	Edit or Delete
Rails Course	No	Edit or Delete
Rails Tutorial	Yes	Edit or Delete

Below the table, there are page navigation links "1 [2](#)". At the bottom of the page is a "Done" button with a checked checkbox next to it.

E em seguida, a segunda página (note o parâmetro adicional da URL):

Terminamos um *controller* completo—simples, mas funcional—e, inclusive, alteramos o modelo de dados por trás do mesmo de acordo com nossa necessidade.

Agora já temos uma base para investigarmos o próximo passo de uma aplicação: relacionamentos entre classes de dados.

RELACIONAMENTOS

Antes de começarmos a ver os relacionamentos, vamos unir o nosso projeto sob um layout que pelo menos nos permita navegar mais facilmente.

Nosso novo arquivo `app/views/layouts/application.rb` ficaria assim:

```
<html>
  <head>
    <title>GTD</title>
    <%= stylesheet_link_tag "default" %>
    <%= stylesheet_link_tag "scaffold" %>
  </head>

  <body>
    <h1 id="header">GTD</h1>
    <ul id="menu">
      <li><%= link_to_unless_current "&raquo; Contexts", :controller => "contexts" %></li>
      <li><%= link_to_unless_current "&raquo; Projects", :controller => "projects" %></li>
    </ul>
    <div id="contents"><%= yield %></div>
  </body>
```

```
</html>
```

E a *stylesheet* que estamos usando, `default.css`, dessa maneira:

```
body
{
    font-family: Verdana, Arial, sans-serif;
    font-size: 100%;
    margin: 0;
    padding: 0;
}

h1#header
{
    background-color: #ccc;
    color: white;
    padding: 10px 10px 15px;
    margin: 0;
}

ul#menu
{
    padding: 5px 10px;
    margin: 0;
    border-bottom: 1px solid #ccc;
}

ul#menu li
{
    display: inline;
    margin-right: 5px;
    font-weight: bold;
}

ul#menu a
{
    text-decoration: none;
    font-weight: normal;
}

div#contents
{
    margin: 10px;
}

div#contents h1
{
    font-size: 130%;
    font-style: italic;
}
```

O resultado final seria:

Name	Active	Actions
Build a House	Yes	Edit or Delete
Compose a Symphony	Yes	Edit or Delete
Plant a Tree	Yes	Edit or Delete
Rails Course	No	Edit or Delete
Rails Tutorial	Yes	Edit or Delete

1 [2](#)

[Done](#)

Não é perfeito por causa de alguns defeitos no código HTML (como o uso de dois elementos H1 na página, tabelas com estilos aplicados diretamente, etc) e pelo uso da *stylesheet scaffold.css*, mas já é uma melhoria (desconsiderando, é claro, o fato de que eu não entendo nada de *design* gráfico). Modificações do primeiro *scaffold* gerado para contextos ficam como um exercício para o leitor. O objetivo é demonstrar como podemos atualizar a aplicação sem mexer em qualquer dos *controllers* e *views* posteriormente geradas uma vez que elas tenham sido bem planejadas.

Para exemplificarmos os relacionamentos, vamos começar com o cadastro de ações. Uma ação pertence a um projeto e é executado em um determinado contexto. Isso é suficiente para começarmos o nosso cadastro.

Atualizamos primeiro o arquivo `application.rhtml` para incluir a nossa nova área, temos:

```
<html>
<head>
  <title>GTD</title>
  <%= stylesheet_link_tag "default" %>
  <%= stylesheet_link_tag "scaffold" %>
```

```

</head>

<body>

  <h1 id="header">GTD</h1>

  <ul id="menu">
    <li><%= link_to_unless_current "&raquo; Contexts", :controller => "contexts", :action => "list"
%></li>
    <li><%= link_to_unless_current "&raquo; Projects", :controller => "projects", :action => "list"
%></li>
    <li><%= link_to_unless_current "&raquo; Actions", :controller => "actions", :action => "list"
%></li>
  </ul>

  <div id="contents"><%= yield %></div>

</body>

</html>

```

Em segundo lugar, geramos o nosso *controller*. Vamos seguir o mesmo estilo do *controller* usado para projetos:

```

ronaldo@minerva:~/tmp/gtd$ script/generate controller actions index list edit delete
exists app/controllers/
exists app/helpers/
create app/views/actions
exists test/functional/
create app/controllers/actions_controller.rb
create test/functional/actions_controller_test.rb
create app/helpers/actions_helper.rb
create app/views/actions/index.rhtml
create app/views/actions/list.rhtml
create app/views/actions/edit.rhtml
create app/views/actions/delete.rhtml

```

Para facilitar, vamos duplicar a funcionalidade presente no cadastro de projetos. O nosso *controller* de ações ficaria assim, inicialmente:

```

class ActionsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @action_pages, @actions = paginate :actions, :per_page => 5, :order => "description"
  end

  def edit
    if params[:id]
      @action = Action.find(params[:id])
    else
      @action = Action.new
    end
    if request.post?
      @action.attributes = params[:action]
      if @action.save
        flash[:notice] = "The action was successfully saved"
        redirect_to :action => "list"
      end
    end
  end
end

```

```

    end
  end
end

def delete
  Action.find(params[:id]).destroy
  flash[:notice] = "The action was successfully deleted"
  redirect_to :action => "list"
end
end

```

Seguido por nossa *view* de listagem, que ficaria assim:

```

<h1>Actions</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New Action", :action => "edit" %></p>

<table border="1" cellpadding="5" cellspacing="1">
  <tr>
    <th>Description</th>
    <th>Done</th>
    <th>Actions</th>
  </tr>
  <% @actions.each do |action| %>
  <tr>
    <td><%= action.description %></td>
    <td><%= yes_or_no?(action.done?) %></td>
    <td>
      <%= link_to "Edit", :action => "edit", :id => action %> or
      <%= link_to "Delete", { :action => "delete", :id => action }, :confirm => "Are you sure?" %>
    </td>
  </tr>
  <% end %>
</table>

<p><%= pagination_links(@action_pages) %></p>

```

Em, por fim, da nossa *view* de edição, que seria:

```

<h1><% if @action.new_record? %>New<% else %>Editing<% end %> Action</h1>

<%= error_messages_for "action" %>

<%= start_form_tag :action => "edit", :id => @action %>

<p>
  <label for="action_description">Description:</label><br>
  <%= text_area "action", "description", :rows => 5 %>
</p>

<p>
  <label for="action_done">Done:</label><br>
  <%= select "action", "done", [ ["Yes", true], [ "No", false ] ] %>
</p>

<p>
  <%= submit_tag "Save" %> or <%= link_to "Cancel", :action => "list" %>
</p>

```

```
<%= end_form_tag %>
```

Não tente usar essa *view* para salvar uma ação ainda. Há um erro sutil na mesma que discutiremos mais adiante. As demais *views* geradas (`index.rhtml` e `delete.rhtml`) podem ser removidas da aplicação pois não serão usadas.

Obviamente, a *view* acima não inclui todos os atributos presentes na ação. Para incluirmos esses atributos, vamos considerar algumas coisas.

O atributo `created_at` representa a data de criação da ação. Atributos terminados em `_at` e `_on` são automaticamente reconhecidos pelo Rails como representando datas e datas e tempos, respectivamente. Além disso, atributos com os nomes `created_at`, `created_on`, `updated_at` e `update_on` serão automaticamente atualizados pelo Rails de acordo com a necessidade. Todos quando o objeto for criado e os dois últimos sempre que o objeto for salvo.

Como o Rails obviamente não tem controle sobre o banco de dados, o desenvolvedor é que deve escolher o tipo de dados certo para os mesmos no banco, usando `date` no caso dos terminados em `_at` e `datetime` no caso dos terminados em `_on` (tipos de dados do MySQL, é claro—adapte conforme a necessidade). Apesar disso, usar um tipo mais ou menos preciso não causa erros e Rails tentará conversões sempre que necessário.

No nosso caso, então, não precisamos dar qualquer valor ao atributo `created_at` para que ele seja salvo. Na verdade, como esse é um campo que deveria ser salvo uma única vez, vamos colocar alguma informação no modelo para que o mesmo não possa ser atribuído em formulários, mas somente em código direto.

```
class Action < ActiveRecord::Base  
  attr_protected :created_at  
end
```

Esse declaração impede que esse atributo seja atualizado automaticamente por qualquer atribuição via formulários ou métodos provindos do servidor. Somente uma atualização direta via código funcionará e como não precisamos disso, não temos que nos preocupar mais com esse atributo. O reverso dessa declaração seria `attr_accessible`.

O atributo `completed_at` representa a data em que a ação foi completada. Também não precisamos editá-lo em nossa *view*, já que ele poderia ser atribuído quando o usuário marcasse a ação como completada. Faremos então a mesma coisa com esse atributo, mudando o nosso modelo para:

```
class Action < ActiveRecord::Base
```

```
attr_protected :created_at, :completed_at  
end
```

Como o campo é atribuído automaticamente, o que precisamos fazer é achar alguma forma de modificar esse atributo quando o atributo `done`, que representa o fato da ação estar completa, for marcado como verdadeiro. A solução é sobreescriver o método de escrita do atributo `done`, algo que o Rails permite justamente para esse tipo de situação:

```
class Action < ActiveRecord::Base  
  
attr_protected :created_at, :completed_at  
  
def done=(value)  
  if ActiveRecord::ConnectionAdapters::Column.value_to_boolean(value)  
    self.completed_at = DateTime.now  
  else  
    self.completed_at = nil  
  end  
  write_attribute("done", value)  
end  
  
end
```

Aqui, estamos usando o momento em que o atributo `done` é recebido e executando ações extras. Note o sinal de igual, indicando que esse é um método de atribuição; no Ruby, qualquer método terminado com o sinal pode ser automaticamente usado como se fosse um atributo. O método verifica se valor recebido foi verdadeiro usando um método interno do `ActiveRecord`. Essa chamada é necessária porque, exceto por `nil` e `false`, qualquer outro valor é considerado verdadeiro em Ruby, incluindo zero e *strings* vazias. Se sim, o atributo `completed_at` é atualizado com a data atual. Se não, o atributo é retornado a um valor nulo. Depois disso, o valor recebido para o atributo `done` é passado sem modificações para a camada de banco de dados, usando o método `write_attribute`.

Mesmo que você não precise modificar campos em dependência um do outro, a técnica acima é útil para guardar valores convertidos (por exemplo, um valor que é informado em graus, mas armazenado em radianos ou vice-versa). O par de métodos `write_attribute/read_attribute` pode ser usado em conjunção com métodos que sobreescrivem os métodos de acesso que o Rails gera para permitir maior flexibilidade da manipulação de dados.

Finalmente, temos os atributos `context_id` e `project_id`, que representam as chaves estrangeiras que criamos no banco, que usaremos para os nossos primeiros relacionamentos entre classes.

BELONGS TO

No caso desses dois atributos, poderíamos dizer que uma ação pertence a um contexto e que pertence a um projeto. No Rails, representaríamos isso da seguinte forma:

```

class Action < ActiveRecord::Base

  attr_protected :created_at, :completed_at

  def done=(value)
    if ActiveRecord::ConnectionAdapters::Column.value_to_boolean(value)
      self.completed_at = DateTime.now
    else
      self.completed_at = nil
    end
    write_attribute("done", value)
  end

  belongs_to :context
  belongs_to :project

end

```

Essas duas declarações são suficientes para criar uma série de facilidades no Rails que inclui atribuições e buscas. Veja que você não precisou dizer nada sobre relacionamento além do que ele referencia. O Rails é capaz de deduzir a chave estrangeira quando ela for criada concatenando o nome da tabela pai ao sufixo `_id`. Caso você tenha dado outro nome, você pode também dizer ao Rails qual seria a chave estrangeira usando algo assim:

```
belongs_to :subject, :foreign_key => "category_id"
```

Além dessa possibilidade, há outras opções na definição de um relacionamento que valem a pena ser exploradas na documentação, incluindo outras condições limítrofes para o mesmo.

Para cada relacionamento criado, o Rails disponibiliza métodos que permitem testar a existência de uma associação, defini-la, removê-la e buscá-la. Vamos usar o console para testar isso:

```

ronaldo@minerva:~/tmp/gtd$ script/console
Loading development environment.

>> action = Action.create(:description => "A test action.")
=> #<Action:0xb745383c @new_record=false, @errors=#<ActiveRecord::Errors:0xb7426490
@base=#<Action:0xb745383c ...>, @errors={}, @attributes={"context_id"=>nil, "completed_at"=>nil,
"project_id"=>nil, "done"=>nil, "id"=>6, "description"=>"A test action.", "created_at"=>Thu Sep 21
15:27:43 BRT 2006}>

```

Temos uma ação já salva.

```

>> action.project_id
=> nil

>> action.context_id
=> nil

>> action.project
=> nil

```

```
>> action.context  
=> nil
```

Com o relacionamento, em adição aos atributos vindos do banco de dados temos agora outros dois atributos que representam a relação.

Podemos atribui-los normalmente:

```
>> action.context_id = 1  
=> 1  
  
>> action.context  
=> #<Context:0xb77aaeb8 @attributes={"name"=>"@Home", "id"=>"1"}>
```

Ou usar diretamente a relação:

```
>> action.project = Project.find(1)  
=> #<Project:0xb77a2da8 @attributes={"name"=>"Build a House", "id"=>"1", "description"=>"Build a  
dream house entirely planned by myself, including a interesting attic with a huge library.",  
"active"=>"1"}>  
  
>> action.project_id  
=> 1
```

Se quisermos, podemos criar também novos valores diretamente:

```
>> action.context = Context.create(:name => "@Phone")  
=> #<Context:0xb779a1d0 @new_record=false, @errors=#<ActiveRecord::Errors:0xb7795a7c  
@base=#<Context:0xb779a1d0 ...>, @errors={}, @attributes={"name"=>"@Phone", "id"=>3}>
```

Relacionamentos são bem úteis, mas precisamos ter um pouco de cuidado em como os usamos. Aqui entra a necessidade de conhecimento de SQL que mencionamos anteriormente. Veja o caso abaixo, por exemplo:

```
>> Action.find(6).project.name  
=> "Build a House"
```

Se executarmos a ação acima e observarmos o *log* de desenvolvimento gerado pelo Rails (que está em *log/development.log*), veremos o seguinte:

```
Action Load (0.002981)  SELECT * FROM actions WHERE (actions.id = 6) LIMIT 1  
Project Load (0.001254)  SELECT * FROM projects WHERE (projects.id = 1) LIMIT 1
```

Isso mostra que duas chamadas foram feitas ao banco para carregar o nome do projeto da ação buscada, uma delas completamente desnecessária. Obviamente, se fizermos isso em um *loop*, com múltiplos objetos e relacionamentos, o nível de ineficiência subirá rapidamente.

Para isso, o Rails tem uma solução que resolve a maior parte dos problemas causados por isso, gerando *joins* automaticamente de acordo com o que o usuário precisa.

Por exemplo:

```
>> Action.find(6, :include => [:project, :context])
=> #<Action:0xb77696f0 @context=#<Context:0xb7768818 @attributes={"name"=>"@Phone", "id"=>"3"}>,
@project=#<Project:0xb7768a48 @attributes={"name"=>"Build a House", "id"=>"1", "description"=>"Build
a dream house entirely planned by myself, including a interesting attic with a huge library.",
"active"=>"1">}, @attributes={"context_id"=>"3", "completed_at"=>nil, "project_id"=>"1", "done"=>nil,
"id"=>"6", "description"=>"A test action.", "created_at"=>"2006-09-21 15:27:43"}>
```

Como você pode ver, tanto o contexto como o projeto foram automaticamente recuperados. Se observamos o *log*, veremos o seguinte:

```
Action Load Including Associations (0.232072)   SELECT actions.`id` AS t0_r0, actions.`description` AS t0_r1, actions.`done` AS t0_r2, actions.`created_at` AS t0_r3, actions.`completed_at` AS t0_r4, actions.`context_id` AS t0_r5, actions.`project_id` AS t0_r6, projects.`id` AS t1_r0, projects.`name` AS t1_r1, projects.`description` AS t1_r2, projects.`active` AS t1_r3, contexts.`id` AS t2_r0, contexts.`name` AS t2_r1 FROM actions LEFT OUTER JOIN projects ON projects.id = actions.project_id LEFT OUTER JOIN contexts ON contexts.id = actions.context_id WHERE (actions.id = 6)
```

Ao invés de duas declaração temos somente uma. Para relacionamentos múltiplos, isso pode reduzir centenas ou milhares de chamadas ao banco em uma única chamada. Veja por exemplo a diferença entre as duas chamadas abaixo, depois de criarmos três ações:

```
>> Action.find(:all).collect { |a| [a.project.name, a.context.name] }
=> [["Build a House", "@Home"], ["Plant a Tree", "@Work"], ["Write a Book", "@Home"]]
```

O objetivo da chamada é retornar uma coleção dos nomes dos projetos e contextos de cada ação existe no banco, em pares. A chamada acima gera os seguintes comandos:

```
Action Load (0.002738)   SELECT * FROM actions
Project Load (0.002161)   SELECT * FROM projects WHERE (projects.id = 1) LIMIT 1
Context Load (0.000834)   SELECT * FROM contexts WHERE (contexts.id = 1) LIMIT 1
Project Load (0.001220)   SELECT * FROM projects WHERE (projects.id = 2) LIMIT 1
Context Load (0.001116)   SELECT * FROM contexts WHERE (contexts.id = 2) LIMIT 1
Project Load (0.001228)   SELECT * FROM projects WHERE (projects.id = 3) LIMIT 1
Context Load (0.001077)   SELECT * FROM contexts WHERE (contexts.id = 1) LIMIT 1
```

Agora, vamos mudar a chamada para:

```
>> Action.find(:all, :include => [:project, :context]).collect { |a| [a.project.name, a.context.name] }
=> [["Build a House", "@Home"], ["Plant a Tree", "@Work"], ["Write a Book", "@Home"]]
```

Veremos, ao executá-la, que somente uma chamada será feita:

```
Action Load Including Associations (0.004515)    SELECT actions.`id` AS t0_r0, actions.`description` AS t0_r1, actions.`done` AS t0_r2, actions.`created_at` AS t0_r3, actions.`completed_at` AS t0_r4, actions.`context_id` AS t0_r5, actions.`project_id` AS t0_r6, projects.`id` AS t1_r0, projects.`name` AS t1_r1, projects.`description` AS t1_r2, projects.`active` AS t1_r3, contexts.`id` AS t2_r0, contexts.`name` AS t2_r1 FROM actions LEFT OUTER JOIN projects ON projects.id = actions.project_id LEFT OUTER JOIN contexts ON contexts.id = actions.context_id
```

É fácil ver que essa chamada é bem melhor do que as demais e isso considerando alguns poucos registros.

Em casos de modelos mais complexos, a partir do Rails 1.1 a declaração *include* é recursiva. Você poderia ter algo como :

```
@orders = Ordem.find :all, :include => [:items => { :product, :discount }, :salesperson]
```

O código acima buscaria todas as ordens de compra presentes em um banco de dados, carregando automaticamente seus itens e a pessoa que a vendeu. Além disso, para os itens, também carregaria os objetos descrevendo o produto associado e o desconto dado. As combinações são infinitas e basta você manter a possibilidade em mente para aplicá-la em suas próprias aplicações.

Vamos agora introduzir nossa modificação final em nosso modelo de dados, as validações:

```
class Action < ActiveRecord::Base

attr_protected :created_at, :completed_at

def done=(value)
  if ActiveRecord::ConnectionAdapters::Column.value_to_boolean(value)
    self.completed_at = DateTime.now
  else
    self.completed_at = nil
  end
  write_attribute("done", value)
end

belongs_to :context
belongs_to :project

validates_presence_of :description

validates_presence_of :context_id
validates_presence_of :project_id

end
```

Agora que temos os nossos relacionamentos, podemos atualizar o nosso *controller* e as *views* geradas para o mesmo. Primeiro, o *controller*:

```
class ActionsController < ApplicationController

def index
  list
  render :action => "list"
end
```

```

def list
  @action_pages, @actions = paginate :actions, :per_page => 5, :order => "description"
end

def edit
  @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
  @projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
  if params[:id]
    @action = Action.find(params[:id])
  else
    @action = Action.new
  end
  if request.post?
    @action.attributes = params[:action]
    if @action.save
      flash[:notice] = "The action was successfully saved"
      redirect_to :action => "list"
    end
  end
end

def delete
  Action.find(params[:id]).destroy
  flash[:notice] = "The action was successfully deleted"
  redirect_to :action => "list"
end

```

As duas linhas acrescentadas criam *arrays* de pares com os nomes e identificadores dos objetos buscados. Precisamos desses *arrays* para a nossa *view* de edição, como vemos abaixo:

```

<h1><% if @action.new_record? %>New<% else %>Editing<% end %> Action</h1>

<%= error_messages_for "action" %>

<%= start_form_tag :action => "edit", :id => @action %>

<p>
  <label for="action_description">Description:</label><br>
  <%= text_area "action", "description", :rows => 5 %>
</p>

<p>
  <label for="action_done">Done:</label><br>
  <%= select "action", "done", [["Yes", true], ["No", false]] %>
</p>

<p>
  <label for="action_context_id">Context:</label><br>
  <%= select "action", "context_id", @contexts, :prompt => "-- Choose --" %>
</p>

<p>
  <label for="action_project_id">Project:</label><br>
  <%= select "action", "project_id", @projects, :prompt => "-- Choose --" %>
</p>

<p>
  <%= submit_tag "Save" %> or <%= link_to "Cancel", :action => "list" %>
</p>

<%= end_form_tag %>

```

Selecionar os objetos e gerar as estruturas que precisamos no *controller* é uma prática recomendada porque mantemos a nossa *view* limpa e podemos fazer quaisquer ordenações e filtros sem nos preocuparmos com o *design* da aplicação. O parâmetro *prompt*, passado acima, permite que um indicador da seleção apareça antes dos valores propriamente ditos.

O formulário resultante fica assim:

The screenshot shows a Firefox browser window titled "GTD - Firefox". The address bar displays "http://localhost:3000/actions/edit". The main content area is a "New Action" form with the following fields:

- Description:** A large text input field.
- Done:** A dropdown menu set to "Yes".
- Context:** A dropdown menu set to "-- Choose --".
- Project:** A dropdown menu set to "-- Choose --".
- Buttons:** "Save" and "Cancel".
- Done:** A button with a checkmark icon.

Agora voltamos ao pequeno problema que indicamos existir nesse controller anteriormente. Se você tentar salvar uma ação agora, verá que um erro acontece:

Action Controller: Exception caught - Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:3000/actions/edit/

NoMethodError in ActionsController#edit

```
undefined method `stringify_keys!' for "edit":String
```

RAILS_ROOT: script/../config/..

[Application Trace](#) | [Framework Trace](#) | [Full Trace](#)

```
/usr/lib/ruby/gems/1.8/gems/activerecord-1.14.4/lib/active_record/base.rb:1506:in `attr'
app/controllers/actions_controller.rb:21:in `edit'
```

Request

Parameters: {"commit"=>"Save"}

[Show session dump](#)

Response

Headers: {"cookie"=>[], "Cache-Control"=>"no-cache"}



Esse erro é causado por um detalhe. No Rails, a chave params[:action] já é tomada pelo nome da ação que está sendo executada e como temos um objeto com esse nome, acabamos com um problema. Entretanto, a solução é simples: basta renomear o nosso objeto. Podemos, por exemplo, chamá-lo de item.

Primeiro, mudamos o controller:

```
class ActionsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @action_pages, @actions = paginate :actions, :per_page => 5, :order => "description"
  end

  def edit
```

```

@contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
@projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
if params[:id]
  @item = Action.find(params[:id])
else
  @item = Action.new
end
if request.post?
  @item.attributes = params[:item]
  if @item.save
    flash[:notice] = "The action was successfully saved"
    redirect_to :action => "list"
  end
end
def delete
  Action.find(params[:id]).destroy
  flash[:notice] = "The action was successfully deleted"
  redirect_to :action => "list"
end
end

```

E depois mudamos a *view*:

```

<h1><% if @item.new_record? %>New<% else %>Editing<% end %> Action</h1>
<%= error_messages_for "item" %>
<%= start_form_tag :action => "edit", :id => @item %>
<p>
  <label for="item_description">Description:</label><br>
  <%= text_area "item", "description", :rows => 5 %>
</p>
<p>
  <label for="item_done">Done:</label><br>
  <%= select "item", "done", [["No", false], ["Yes", true]] %>
</p>
<p>
  <label for="item_context_id">Context:</label><br>
  <%= select "item", "context_id", @contexts, :prompt => "-- Choose --" %>
</p>
<p>
  <label for="item_project_id">Project:</label><br>
  <%= select "item", "project_id", @projects, :prompt => "-- Choose --" %>
</p>
<p>
  <%= submit_tag "Save" %> or <%= link_to "Cancel", :action => "list" %>
</p>
<%= end_form_tag %>

```

Esse é um caso raro, mas se você vir algum erro misterioso acontecendo do Rails onde tudo deveria estar funcionando, verifique se não há um conflito.

Como um detalhe extra, invertemos acima a ordem do *Yes* e *No* na seleção da situação ativa para o correto que seriam uma ação ainda não concluída por padrão. Rodando agora a ação, temos o resultado desejado.

Vamos agora modificar o *controller* mais uma vez em apoio à *view* de listagem dos dados. Teríamos o seguinte:

```
class ActionsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @action_pages, @actions = paginate :actions, :include => [:project, :context],
    :per_page => 5, :order => "actions.description"
  end

  def edit
    @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    @projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    if params[:id]
      @item = Action.find(params[:id])
    else
      @item = Action.new
    end
    if request.post?
      @item.attributes = params[:item]
      if @item.save
        flash[:notice] = "The action was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def delete
    Action.find(params[:id]).destroy
    flash[:notice] = "The action was successfully deleted"
    redirect_to :action => "list"
  end
end
```

A inclusão dos relacionamentos irá melhorar a nossa *view*. Veja que precisamos agora especificar a condição de ordenação mais claramente, já que temos uma coluna chamada `description` em duas tabelas.

Nossa *view* ficaria assim:

```
<h1>Actions</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New Action", :action => "edit" %></p>

<table border="1" cellpadding="5" cellspacing="1">
  <tr>
    <th>Description</th>
    <th>Completed</th>
    <th>Project</th>
    <th>Context</th>
    <th>Actions</th>
  </tr>
  <% @actions.each do |action| %>
```

```

<tr>
  <td><%= action.description %></td>
  <td><%= yes_or_no?(action.done?) %></td>
  <td><%= action.project.name %></td>
  <td><%= action.context.name %></td>
  <td>
    <%= link_to "Edit", :action => "edit", :id => action %> or
    <%= link_to "Delete", { :action => "delete", :id => action }, :confirm => "Are you sure?" %>
  </td>
</tr>
<% end %>
</table>

<p><%= pagination_links(@action_pages) %></p>

```

Dando o seguinte resultado:

Description	Completed	Project	Context	Actions
Get permission from the powers that be to plant a tree in the lounge.	No	Plant a Tree	@Work	Edit or Delete
Sit down and write, every day, at least two hours a day.	No	Write a Book	@Home	Edit or Delete

Se quisermos, podemos também acrescentar um campo informando quando a ação foi completada, formatando a data em que a mesma foi completada da maneira necessária à nossa aplicação, ou exibindo um marcador caso contrário. Nossa *view* poderia ficar como algo assim:

```

<h1>Actions</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New Action", :action => "edit" %></p>

```

```


| Description               | Completed                       | When                                                                            | Project                        | Context                        | Actions                                                                                                                                                                            |
|---------------------------|---------------------------------|---------------------------------------------------------------------------------|--------------------------------|--------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <%= action.description %> | <%= yes_or_no?(action.done?) %> | <td><%= (action.done?) ? action.completed_at.strftime("%m/%d/%y") : "-" %></td> | <td><%= action.project.name %> | <td><%= action.context.name %> | <td>       <%= link_to "Edit", :action => "edit", :id => action %> or       <%= link_to "Delete", { :action => "delete", :id => action }, :confirm => "Are you sure?" %>     </td> |
| <% end %>                 |                                 |                                                                                 |                                |                                |                                                                                                                                                                                    |



<p><%= pagination_links(@action_pages) %></p>


```

Com o seguinte resultado:

Description	Completed	When	Project	Context	Actions
Get permission from the powers that be to plant a tree in the lounge.	Yes	09/21/06	Plant a Tree	@Work	Edit or Delete
Sit down and write, every day, at least two hours a day.	No	-	Write a Book	@Home	Edit or Delete

Done

HAS MANY

Além do relacionamento `belongs_to`, o Rails também possui um outro relacionamento chamado `has_many`. Como o próprio nome indica, esse relacionamento diz que um modelo possui muitos de outros modelos. Em nossa aplicação, nos temos duas instâncias imediatas disso: um contexto possui muitas ações e um projeto possui muitas ações.

Modificando inicialmente o modelo de contextos, teríamos:

```
class Context < ActiveRecord::Base  
  
  validates_presence_of :name  
  validates_uniqueness_of :name, :message => "must be unique"  
  
  has_many :actions  
  
end
```

Isso é suficiente para estabelecer uma série de métodos adicionais que permitem que você manipule as ações relacionadas a um contexto. Usando o console, podemos ver algumas delas:

```
ronaldo@minerva:~/tmp/gtd$ script/console  
Loading development environment.  
  
>> context = Context.find(1)  
=> #<Context:0xb73ff1f8 @attributes={"name"=>"@Home", "id"=>"1"}>  
  
>> context.actions.count  
=> 1  
  
>> context.actions  
=> [#<Action:0xb73f296c @attributes={"context_id"=>"1", "completed_at"=>nil, "project_id"=>"3",  
"done"=>"0", "id"=>"9", "description"=>"Sit down and write, every day, at least two hours a day.",  
"created_at"=>"2006-09-21 16:31:14"}>]
```

Se criarmos mais uma ação associada a esse contexto, teríamos algo assim:

```
>> action = Action.create(:description => "Buy a grammar to help me with my English.", :context_id =>  
1, :project_id => 3)  
=> #<Action:0xb78f5448 @new_record=false, @errors=#<ActiveRecord::Errors:0xb78e8644  
@base=#<Action:0xb78f5448 ...>, @errors={}, @attributes={"context_id"=>1, "completed_at"=>nil,  
"project_id"=>3, "done"=>nil, "id"=>11, "description"=>"Buy a grammar to help me with my English.",  
"created_at"=>Thu Sep 21 16:52:40 BRT 2006}>  
  
>> context.actions  
=> [#<Action:0xb73f296c @attributes={"context_id"=>"1", "completed_at"=>nil, "project_id"=>"3",  
"done"=>"0", "id"=>"9", "description"=>"Sit down and write, every day, at least two hours a day.",  
"created_at"=>"2006-09-21 16:31:14"}>]  
  
>> context.actions.reload  
=> [#<Action:0xb77786cc @attributes={"context_id"=>"1", "completed_at"=>nil, "project_id"=>"3",  
"done"=>"0", "id"=>"9", "description"=>"Sit down and write, every day, at least two hours a day.",  
"created_at"=>"2006-09-21 16:31:14"}, #<Action:0xb7778690 @attributes={"context_id"=>"1",  
"completed_at"=>nil, "project_id"=>"3", "done"=>nil, "id"=>"11", "description"=>"Buy a grammar to  
help me with my English.", "created_at"=>"2006-09-21 16:52:40"}>]
```

```
>> context.actions.count  
=> 2  
  
>> context.actions.empty?  
=> false
```

Note que, como usando um objeto já carregado, a coleção de ações não foi recarregada depois que uma nova ação foi adicionada. Isso acontece porque o Rails faz um *cache* de coleções e a ação foi adicionada através de seu próprio *constructor* e não usando os métodos da coleção. O método `reload` pode ser usado para recarregar a coleção caso você precise.

Um outro exemplo seria:

```
>> context.actions << Action.create(:description => "Upgrade my word processor.", :project_id => 3)  
=> [#<Action:0xb77786cc @attributes={"context_id"=>"1", "completed_at"=>nil, "project_id"=>"3",  
"done"=>"0", "id"=>"9", "description"=>"Sit down and write, every day, at least two hours a day.",  
"created_at"=>"2006-09-21 16:31:14"}], #<Action:0xb7778690 @attributes={"context_id"=>"1",  
"completed_at"=>nil, "project_id"=>"3", "done"=>nil, "id"=>"11", "description"=>"Buy a grammar to  
help me with my English.", "created_at"=>"2006-09-21 16:52:40"}], #<Action:0xb7753160  
@new_record=false, @errors=#<ActiveRecord::Errors:0xb77513b0 @base=#<Action:0xb7753160 ...>,  
@errors={}, @attributes={"context_id"=>1, "completed_at"=>nil, "project_id"=>3, "done"=>nil,  
"id"=>12, "description"=>"Upgrade my word processor.", "created_at"=>Thu Sep 21 16:59:21 BRT 2006}]  
  
>> context.actions.size  
=> 3
```

Nesse caso, você pode ver que duas coisas aconteceram: primeiro, não foi necessário informar o contexto, que é pego automaticamente no objeto; segundo: a coleção cresceu automaticamente, já que estamos manipulando os dados diretamente na mesma.

Ao invés de usar os métodos acima para criar a ação, você pode usar diretamente os métodos `build` e `create` na coleção para adicionar novos dados. O primeiro funciona para objetos já existentes na coleção e o segundo para adicionar um novo objeto.

Uma coleção também possui um método `find` para encontrar objetos dentro da mesma, de acordo com as mesmas regras do método `find` usado para classes de dados.

Por fim, por razões de eficiência, toda coleção declara um método do objeto pai para atribuição rápida de valores. No caso do objeto acima, o método é chamada `action_ids` e pode ser usado para trocar completamente os dados de uma coleção. Por exemplo:

```
context.action_ids = [1, 2, 6]
```

Isso removeria qualquer ação associada àquele contexto, substituindo-as pelas ações identificadas pelos *ids* acima.

A coleção toda pode ser substituída com objetos também, como mostrado abaixo:

```
context.actions = Action.find(:all, :conditions => ["project_id = ?", 5])
```

O contexto acima teria todas as suas ações substituídas pelas ações associadas ao projeto cujo *id* é 5.

Investigando a documentação você poderá encontrar mais detalhes sobre como cada método funciona. Um exemplo disso é a destruição automática de registros quando um registro pai for excluído. Isso pode fazer um objeto ter a funcionalidade equivalente de uma declaração *cascade* em um banco de dados.

Uma coisa que deve ser notada é que muitos desses métodos causam o salvamento imediato dos dados (se o objeto pai já está salvo). Com a prática, você será capaz de identificar quais são esses métodos e agir apropriadamente caso queria outra ação, embora o comportamento padrão seja o desejado na maior parte dos casos.

Não vamos utilizar esses métodos em nossa aplicação no momento porque veremos uma aplicação similar mais à frente com outro tipo de relacionamento parecido.

Has one

Dois outros relacionamentos existentes no Rails são `has_and_belongs_many` e `has_one`.

O relacionamento `has_one` geralmente expressa o oposto do relacionamento `belongs_to`. Por exemplo, digamos que temos duas tabelas: uma contendo cartões de crédito e outra contendo os titulares de cartões. Na tabela de cartão de crédito teríamos uma chave estrangeira apontando para o seu titular. Teríamos então as seguintes classes:

```
class CreditCard < ActiveRecord::Base
  belongs_to :account_holder
end

class AccountHolder < ActiveRecord::Base
  has_one :credit_card
end
```

A diferença é que a chave estrangeira existe somente na tabela relacionada à classe `CreditCard`, sendo automaticamente deduzida no relacionamento inverso.

Esse exemplo não é muito bom já que, em tese, uma pessoa poderia ter vários cartões de crédito. Mas a ideia é essa. Na prática, esse tipo de relacionamento é mais raro e geralmente tende a se transformar em relacionamentos do tipo `has_many` ou `has_many_and_belongs_to`.

O relacionamento `has_many_and_belongs_to`, por sua vez, representa a intermediação entre duas tabelas.

Por exemplo, a relação entre desenvolvedores e projetos em uma empresa. Um desenvolvedor pode estar locado em múltiplos projetos e um projeto pode ter múltiplos desenvolvedores. Veremos mais detalhes desse relacionamento adiante quando fizermos mais modificações em nossa aplicação para suportar outras características.

HAS MANY, THROUGH

Um tipo interessante de relacionamento, introduzido no Rails 1.1, é aquele relacionamento em que uma tabela intermediária é usada para colapsar outra. Digamos, por exemplo, que você tenha modelos representando empresas, clientes e notas fiscais e você queira descobrir todas as notas fiscais emitidas por um empresa qualquer. É aqui que esse tipo de relacionamento entra em ação, permitindo que o código seja simplificado. Vamos experimentar um pouco, usando o console.

Digamos que você queria, em nossa aplicação, saber todos os contextos associados a um projeto. A maneira simples seria fazer isso:

```
ronaldo@minerva:~/tmp/gtd$ script/console
Loading development environment.

>> project = Project.find(3)
=> #<Project:0xb78083fc @attributes={"name"=>"Write a Book", "id"=>"3", "description"=>"", "active"=>"1"}>

>> project.actions.collect { |a| a.context.name }
=> ["@Home", "@Home", "@Home"]
```

Como você pode ver, o contexto é o mesmo para todas as ações associadas àquele projeto e queremos os contextos sem repetição. Uma solução simples seria:

```
>> contexts = project.actions.collect { |a| a.context.name }
=> ["@Home", "@Home", "@Home"]

>> contexts.uniq
=> ["@Home"]
```

O problema é que, além de usar mais código, essa solução é muito inefficiente por causa das comparações que precisam ser realizadas pelo método `uniq`. Podemos resolver o problema usando então a seguinte estratégia:

```
class Project < ActiveRecord::Base

  validates_presence_of :name
  validates_uniqueness_of :name

  has_many :actions
  has_many :contexts, :through => :actions, :select => "distinct contexts.*"

end
```

Usando esse método temos:

```
>> project.contexts.collect { |c| c.name }
=> ["@Home"]
```

Como você pode ver, muito mais fácil e interessante e, em quase todos os casos, mais eficiente.

No relacionamento acima, temos os contextos de um projeto obtidos através (*through*) de suas ações. Se não for necessário preocupar com cópias (como no exemplo das notas fiscais) poderemos inclusive descartar o parâmetro *select* que é um fragmento de SQL usado no lugar do gerado automaticamente pelo Rails.

Associações *through* são muito poderosas e podem ser utilizadas em várias situações para agilizar o código e melhorar a legibilidade da aplicação

Mais adiante, veremos também outra extensão de associações *has_many* conhecidas como associações polimórficas, que também pode ser usadas para mais combinações interessantes de modelos de dados.

Agora que já vimos o básico para um aplicação, é hora de focar em alguns tópicos mais avançados.

Avançando no Rails

FILTROS

Nossa aplicação até o momento estaria disponível para qualquer pessoa que desejasse usá-la. Em um cenário normal, provavelmente gostaríamos de restringir acesso para somente um grupo de usuários permitidos.

Vamos, então, adicionar um sistema rápido de autenticação à aplicação, aproveitando para explorar mais alguns conceitos do Rails. Uma observação aqui é que existem *plugins* prontos com sistemas de autenticação funcionais, plenamente customizáveis. Como tudo no Rails, a escolha é sua.

A primeira coisa que temos que fazer é criar uma tabela de usuários para conter os *logins* e senhas das pessoas que poderão acessar o sistema. Para isso, vamos seguir o caminho familiar, gerando um *model* para representar a nossa tabela:

```
ronaldo@minerva:~/tmp/gtd$ script/generate model user
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/user.rb
create  test/unit/user_test.rb
create  test/fixtures/users.yml
```

```
exists  db/migrate
create  db/migrate/005_create_users.rb
```

Editando a migração gerada, ficaríamos, pensando em algo bem básico, com o seguinte:

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.column :name, :string
      t.column :login, :string, :limit => 10
      t.column :password, :string, :limit => 10
    end
    User.reset_column_information
    User.create(:name => "Ronaldo", :login => "ronaldo", :password => "test")
  end

  def self.down
    drop_table :users
  end
end
```

Note o uso do parâmetro *limit* para gerar campos texto com um tamanho menor e que já estou criando um usuário inicial para podermos efetuar autenticação logo depois de termos modificado a aplicação sem precisarmos de criar um cadastro de usuários imediatamente. Esse cadastro fica como um exercício para o leitor, sendo uma boa oportunidade para usar validações como `validates_confirmation_of`, que você pode achar na documentação do Rails.

O que precisamos agora é alguma maneira de bloquear o acesso aos nossos *controllers* já criados caso o usuário não esteja autenticado. Isso é conseguido com algo que o Rails chama de filtros, que são métodos executados antes ou depois do processamento de uma ação. O Rails possui três tipos de filtros: os executados antes de uma ação, os executados depois de uma ação e os executados antes e depois de uma ação (*around filters*, na terminologia da documentação).

Qualquer filtro em Rails pode retornar um valor falso para bloquear o processamento posterior da ação. No nosso caso, é exatamente disso que precisamos: caso um usuário não esteja autenticado, não permitiremos que ele acesse um *controller*.

Como queremos fazer isso para todos os nossos *controllers* existentes, usaremos um filtro no *controller* do qual todos os outros são derivados: o `ApplicationController`, já mencionado anteriormente. E para saber se o usuário está ou não autenticado, usaremos uma variável de sessão.

Modificando o nosso código, inicialmente teríamos algo assim:

```
class ApplicationController < ActionController::Base
  before_filter :authenticate
  protected
```

```

def authenticate
  unless session[:user]
    false
  end
end

end

```

No código acima, a menos que tenhamos uma variável de sessão chamada `session[:user]`, retornaremos falso, bloqueando o processamento da ação. Se você rodar qualquer *controller* agora, verá que uma tela completamente vazia será retornada, sem nem mesmo o código do *layout*.

Obviamente, o que precisamos é de uma forma de permitir que o usuário faça sua autenticação. Vamos modificar o código acima para lidar com essa possibilidade e enviar o usuário para um local onde ele possa fazer autenticação caso ainda não tenha feito:

```

class ApplicationController < ActionController::Base

  before_filter :authenticate

  protected

    def authenticate
      unless session[:user]
        session[:return_to] = request.request_uri
        redirect_to :controller => "login", :action => "login"
        return false
      end
      return true
    end

  end

```

O nosso código agora verifica se existe uma variável de sessão que precisamos. Se a variável não existir, redirecionamos o usuário para um novo *controller* que fará a sua autenticação. Nesse caso, retornamos falso para bloquear o resto da ação, seja qual for. Se a variável existir, simplesmente retornamos verdadeiro para deixar que o código siga seu caminho usual. Um toque adicional é salvar a URL que o usuário tentou acessar para retornar a ela uma vez que a autenticação seja feita.

Agora, precisamos de nosso *controller* de autenticação. E além da autenticação, podemos também permitir que o usuário encerre sua sessão, para que ela não fique aberta depois que ele saiu da aplicação. Sendo assim, precisamos de duas ações: *login* e *logout*. De acordo com isso, geramos o *controller* abaixo:

```

ronaldo@minerva:~/tmp/gtd$ script/generate controller login login logout
exists  app/controllers/
exists  app/helpers/
create  app/views/login
exists  test/functional/
create  app/controllers/login_controller.rb
create  test/functional/login_controller_test.rb
create  app/helpers/login_helper.rb
create  app/views/login/login.rhtml
create  app/views/login/logout.rhtml

```

Se você rodar uma página qualquer agora verá que o servidor entra um *loop* infinito. Isso acontece por causa de um detalhe fundamental: o *controller* que acabamos de criar também está sujeito ao filtro. A solução é simples. Basta adicionar uma linha ao *controller*:

```
class LoginController < ApplicationController
  skip_before_filter :authenticate, :only => [:login]
  def login
  end
  def logout
  end
end
```

A linha acima instrui o *controller* a ignorar o filtro de autenticação, somente para a ação *login*. Essa é uma boa prática de segurança, fazendo exclusões somente quando necessário. No caso acima, mesmo para a ação *logout* teríamos que estar autenticados. Na nossa implementação é provável que essa ação não faça nada de mais. Em um implementação real, porém, ela poderia ser tão importante quanto qualquer outra ação.

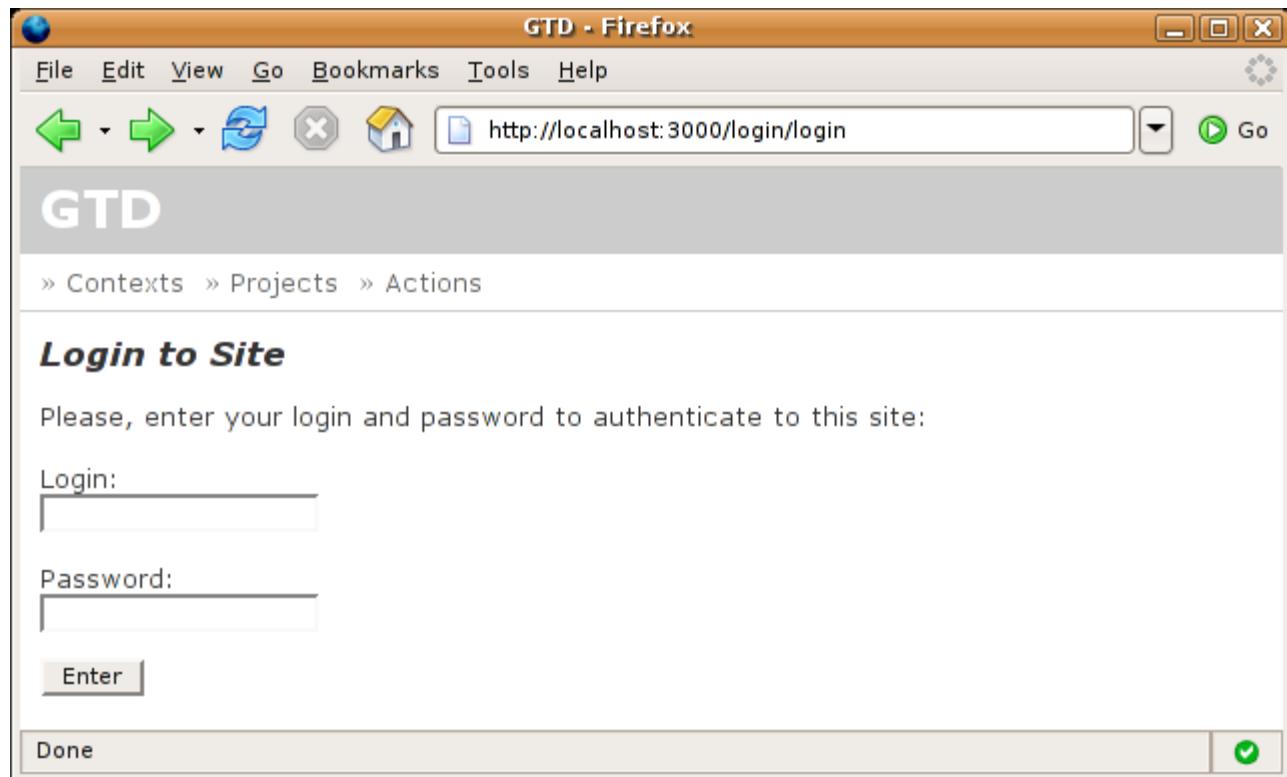
Resolvido esse problema, podemos partir para a nossa implementação da autenticação. Para que o usuário seja capaz de fazer sua autenticação, precisamos de um formulário. Para isso vamos alterar a *view* responsável pela autenticação, *app/views/login/login.rhtml*:

```
<h1>Login to Site</h1>
<% if flash[:notice] %>
<p style="color: red; font-style: italic"><%= flash[:notice] %></p>
<% end %>
<%= start_form_tag :action => "login" %>
<p>Please, enter your login and password to authenticate to this site:</p>
<p>
  <label for="login">Login:</label><br>
  <%= text_field_tag "login", params[:login], :maxlength => 10 %>
</p>
<p>
  <label for="password">Password:</label><br>
  <%= password_field_tag "password", "", :maxlength => 10 %>
</p>
<p>
  <%= submit_tag "Enter" %>
</p>
<%= end_form_tag %>
```

Veja que estamos usando versões diferentes dos métodos de geração de campos de formulários. Isso acontece porque não precisamos de atribuir dados a um objeto, mas simplesmente recolher os dados de autenticação. Como esses métodos não recebem dados de um objeto, o segundo parâmetro dos mesmos é

um valor a ser inicializado. No caso, usamos um *login* previamente submetido para preencher o campo de *login* caso ele exista, e sempre deixamos a senha em branco depois de uma falha de autenticação. Já preparamos também uma forma de avisar ao usuário em casos de falhas de autenticação, usando variáveis *flash*.

Teríamos o seguinte resultado:



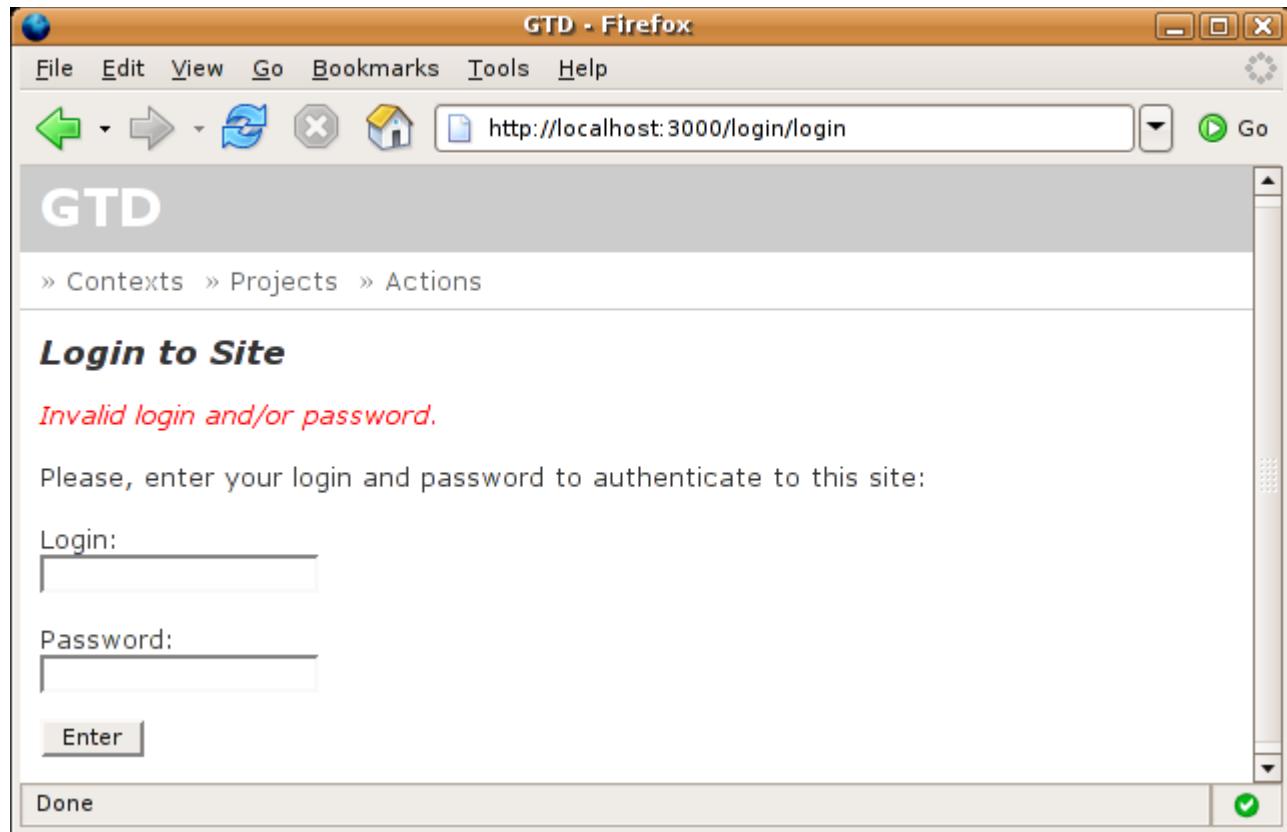
Precisamos agora modificar o nosso *controller* para realizar a autenticação:

```
class LoginController < ApplicationController
  skip_before_filter :authenticate, :only => [:login]

  def login
    if request.post?
      @user = User.find_by_login_and_password(params[:login], params[:password])
      if @user
        session[:user] = @user.id
        if session[:return_to] && !session[:return_to].include?(url_for(:action => "login"))
          redirect_to session[:return_to]
          session[:return_to] = nil
        else
          redirect_to :controller => "home"
        end
      else
        flash[:notice] = "Invalid login and/or password."
      end
    end
  end
end
```

```
def logout
end
end
```

O resultado ao tentarmos efetuar um *login* inválido seria o seguinte:



Um *login* válido nos levaria para o *controller* que tentamos acessar, seja qual for ele. O código é bem simples.

Analisando linha a linha, teríamos uma seqüência fácil. Primeiro, testamos se o formulário está sendo submetido. Se estiver, tentamos encontrar um usuário que atenda àquele *login* e senha. Aqui você pode ver um uso dos métodos `find_*` automaticamente gerados.

Se não encontrarmos um registro que atenda esses critérios, simplesmente renderizamos a mesma página, informando uma mensagem ao usuário.

Caso encontremos um usuário, porém, registramos o seu *id* na variável de sessão que escolhemos. Os motivos para registrar somente o *id* são duplos: um, é muito mais eficiente armazenar um número do que um objeto completo que teria que ser montado a cada requisição da sessão; dois, se a estrutura do objeto mudar ou, por algum motivo, o Rails não encontrar a classe naquela requisição, um erro será gerado.

Após registramos o *id* do usuário, verificamos se há algum lugar para redirecionarmos. Testamos se existe ou não um redirecionamento e, caso exista, se não é para o *controller* que estamos agora. Isso, mais uma vez, evita erros e redireções infinitas. Se tivermos um valor válido, redirecionamos para o mesmo. Se não, meramente acessamos a página inicial da aplicação.

Com isso, temos um sistema simples e funcional de autenticação que você pode aumentar de acordo com suas necessidades, adicionando autenticação segura, autenticação via *hashes* ou qualquer outra coisa que deseje. Notamos mais uma vez que existem *plugins* para ajudar na implementação dessas tarefas.

Podemos aproveitar a informação que temos de sessão e modificar o nosso *layout* para dar mais uma facilidade ao usuário. Primeiro, precisamos de um *helper* para nos dar o usuário autenticado no momento. Podemos colocar isso no *helper* global da aplicação, modificando o arquivo `application_helpers.rb`:

```
module ApplicationHelper

  def yes_or_no?(value)
    value ? "Yes" : "No"
  end

  def session_user
    @session_user ||= User.find(:first, :conditions => ['id = ?', session[:user]])
  end

end
```

A técnica acima é um demonstração do que há de melhor do Ruby. Queremos retornar o objeto associado ao usuário autenticado. Pode acontecer de termos que usar esse objeto mais de um vez. Fazer um requisição no banco toda vez em que precisarmos do objeto seria bem ineficiente.

O que fazemos então é criar uma variável de instância que contenha o objeto retornado. A sintaxe acima significa que retornamos a variável de instância ou (`||=`) o valor da chamada ao método `find` da classe `User`, o primeiro que não for nulo. Como da primeira vez que a chamada for feita a variável de sessão será nula, o retorno será o da chamada. Mas, nesse momento, atribuímos (`||=`) o valor retornado à variável, que será então retornado em todas as vezes subsequentes.

Note que usamos `find` com `:first` para evitarmos um erro caso o usuário não exista. Nesse caso, queremos somente que um valor nulo seja dado quando a sessão não existir, ao contrário de um erro.

Agora, modificamos o *layout* da aplicação, em `app/views/layouts/application.rhtml`:

```
<html>
<head>
  <title>GTD</title>
  <%= stylesheet_link_tag "default" %>
```

```

<%= stylesheet_link_tag "scaffold" %>
</head>

<body>

  <h1 id="header">GTD</h1>

  <% if session_user %>
  <p id="login-information">
    You are logged in as <strong><%= session_user.name %></strong>.
    <%= link_to "Log out", :controller => "login", :action => "logout" %>.
  </p>
  <% end %>

  <ul id="menu">
    <li><%= link_to_unless_current "&raquo; Contexts", :controller => "contexts", :action => "list"
    %></li>
    <li><%= link_to_unless_current "&raquo; Projects", :controller => "projects", :action => "list"
    %></li>
    <li><%= link_to_unless_current "&raquo; Actions", :controller => "actions", :action => "list"
    %></li>
  </ul>

  <div id="contents"><%= yield %></div>

</body>

</html>

```

Por fim, acrescentamos o seguinte fragmento de estilo à *stylesheet* da aplicação em `public/stylesheets/default.css`:

```

#login-information
{
  position: absolute;
  right: 10px;
  top: 5px;
  margin: 0;
  padding: 0;
  font-size: small;
}

```

O resultado é esse:

Podemos agora implementar a ação que remove a sessão do usuário, que é bem simples:

```
class LoginController < ApplicationController

  skip_before_filter :authenticate, :only => [:login]

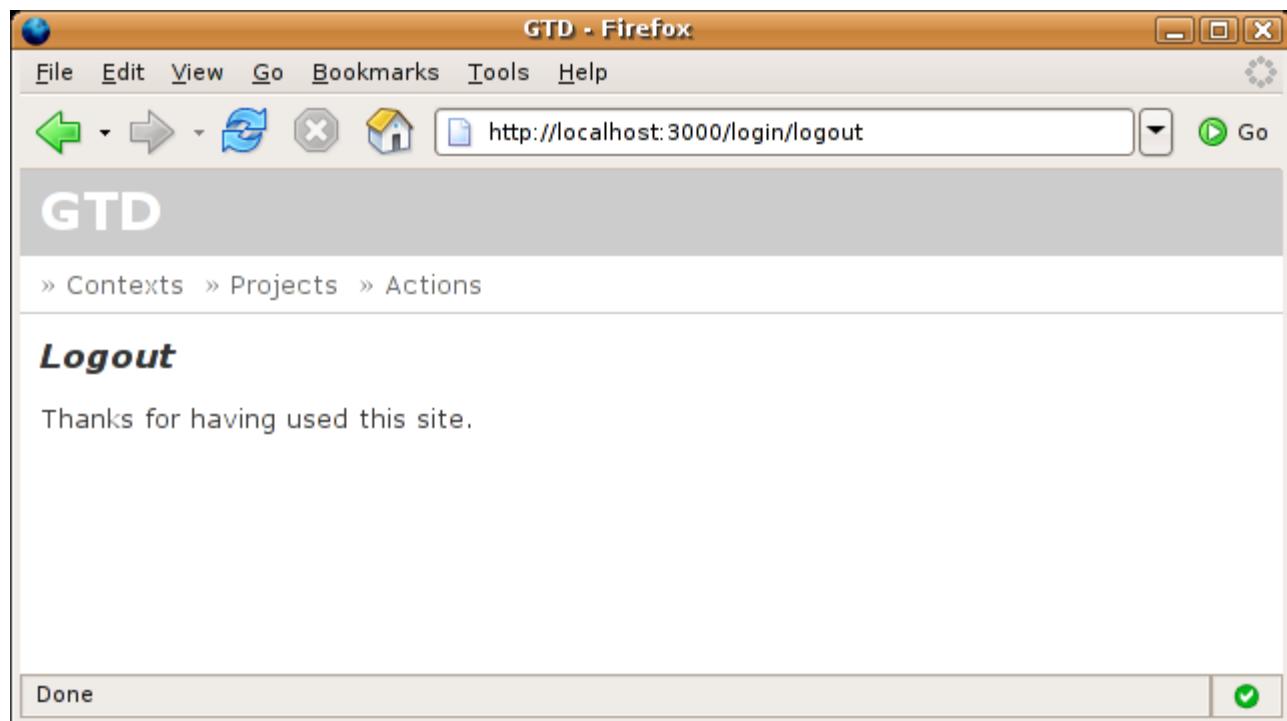
  def login
    if request.post?
      @user = User.find_by_login_and_password(params[:login], params[:password])
      if @user
        session[:user] = @user.id
        if session[:return_to] && !session[:return_to].include?(url_for(:action => "login"))
          redirect_to session[:return_to]
          session[:return_to] = nil
        else
          redirect_to :controller => "home"
        end
      else
        flash[:notice] = "Invalid login and/or password."
      end
    end
  end

  def logout
    session[:user] = nil
  end
end
```

Modificamos também nossa view (app/views/login/logout.rhtml):

```
<h1>Logout</h1>
<p>Thanks for having used this site.</p>
```

E o resultado de sair da aplicação é:



Filtros também podem ser usados em múltiplas combinações. Para demonstrar isso, vamos expandir o modelo de dados de usuários para incluir também o conceito de administradores do sistema, que serão os únicos que podem cadastrar novos usuários.

Começamos gerando uma migração:

```
ronaldo@minerva:~/tmp/gtd$ script/generate migration add_administrative_users
      create db/migrate
      create db/migrate/006_add_administrative_users.rb
```

E editando a mesma:

```
class AddAdministrativeUsers < ActiveRecord::Migration
  def self.up
    add_column :users, :admin, :boolean
    User.reset_column_information
    User.find_by_name("ronaldo").toggle!(:admin)
  end

  def self.down
    remove_column :users, :admin
  end
end
```

Como não temos até agora uma administração de usuários, podemos criá-la, seguindo o padrão dos outros

controllers que temos. Se você gerou esse *controller* anteriormente, o resultado final deve ser assemelhar a algo assim:

The screenshot shows a Firefox browser window titled "GTD - Firefox". The address bar displays "http://localhost:3000/users/list". The page content is as follows:

You are logged in as **Ronaldo**. [Log out.](#)

» Contexts » Projects » Actions » Resources » **Users**

Users

[New User](#)

Name	Login	Administrator	Actions
Marcus	marcus	No	Edit or Delete
Ronaldo	ronaldo	Yes	Edit or Delete

Done

Lembre-se de alterar o arquivo `application.rhtml` em `app/views/layouts` para refletir o novo menu.

Como a área de usuário somente pode ser acessar por usuários administrativos, precisamos de uma forma de recusar acesso à mesma caso o usuário não tenha esse perfil.

Faremos isso com o uso de um filtro extra, aplicável somente aos *controllers* que estiverem sob administração, que, no momento, é somente o próprio *controller* de usuários. Entretanto, como o filtro pode ser aplicado a mais de um *controller* deixaremos sua definição no *controller* raiz, `application.rb`:

```
class ApplicationController < ActionController::Base
  before_filter :authenticate
  protected
    def authenticate
      unless session[:user]
        session[:return_to] = request.request_uri
        redirect_to :controller => "login", :action => "login"
        return false
      end
      return true
    end
    def authenticate_administration
      unless session[:user] && session[:user].admin?
        redirect_to :controller => "login", :action => "login"
        return false
      end
    end
end
```

```

    redirect_to :action => "access_denied"
    return false
  end
  return true
end

end

```

A idéia é simples: se houver acesso administrativo, deixamos a ação continuar; caso contrário, redirecionamos para uma ação que informa que o acesso foi negado. Agora, precisamos somente aplicar o filtro ao *controller* de usuários:

```

class UsersController < ApplicationController

  before_filter :authenticate_administration

  def index
    list
    render :action => "list"
  end

  def list
    @user_pages, @users = paginate :users, :per_page => 5, :order => "name"
  end

  def edit
    if params[:id]
      @user = User.find(params[:id])
    else
      @user = User.new
    end
    if request.post?
      @user.attributes = params[:user]
      if @user.save
        flash[:notice] = "The user was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def delete
    User.find(params[:id]).destroy
    flash[:notice] = "The user was successfully deleted"
    redirect_to :action => "list"
  end
end

```

Ao carregarmos o *controller* de usuário, veremos que um erro acontece, já que o método `session_user` não existe nem nesse *controller* nem no *controller* raiz, estando presente somente no *helper*. A solução é dada pelo próprio Rails: movemos o método do *helper* para o *controller* e o declaramos lá como um *helper*, mantendo o acesso ao mesmo nas *views*. O resultado final, depois de movermos o método é o seguinte:

```

class ApplicationController < ActionController::Base

  before_filter :authenticate

  helper_method :session_user

  protected

```

```

def session_user
  @session_user ||= User.find(:first, :conditions => ['id = ?', session[:user]])
end

def authenticate
  unless session[:user]
    session[:return_to] = request.request_uri
    redirect_to :controller => "login", :action => "login"
    return false
  end
  return true
end

def authenticate_administration
  unless session_user && session_user.admin?
    redirect_to :action => "access_denied"
    return false
  end
  return true
end

end

```

Precisamos agora somente de uma modificação em nosso arquivo de *layout* para refletir os cadastros que não estão acessíveis a usuários comuns:

```

<html>

<head>
  <title>GTD</title>
  <%= stylesheet_link_tag "default" %>
  <%= stylesheet_link_tag "scaffold" %>
</head>

<body>

  <h1 id="header">GTD</h1>

  <% if session_user %>
  <p id="login-information">
    You are logged in as <strong><%= session_user.name %></strong>.
    <%= link_to "Log out", :controller => "login", :action => "logout" %>.
  </p>
  <% end %>

  <ul id="menu">
    <li><%= link_to_unless_current "&raquo; Contexts", :controller => "contexts", :action => "list"
    %></li>
    <li><%= link_to_unless_current "&raquo; Projects", :controller => "projects", :action => "list"
    %></li>
    <li><%= link_to_unless_current "&raquo; Actions", :controller => "actions", :action => "list"
    %></li>
    <li><%= link_to_unless_current "&raquo; Resources", :controller => "resources", :action => "list"
    %></li>
    <% if session_user && session_user.admin? %>
      <li><%= link_to_unless_current "&raquo; Users", :controller => "users", :action => "list" %></li>
    <% end %>
  </ul>

  <div id="contents"><%= yield %></div>

</body>

</html>

```

Acessando com um usuário administrador, teríamos:

The screenshot shows a Firefox browser window with the title bar "GTD - Firefox". The address bar displays the URL "http://localhost:3000/contexts/list". The main content area is titled "GTD" and shows a breadcrumb navigation path: "» Contexts » Projects » Actions » Resources » Users". Below this, the heading "Contexts" is displayed. A link "New Context" is visible. A table lists contexts with columns "Name" and "Actions". The table contains two rows: one for "@Home" with actions "Edit or Delete", and one for "@Work" with actions "Edit or Delete". At the bottom of the page is a "Done" button with a checked checkbox.

E com um usuário que não seja administrador:

The screenshot shows a Firefox browser window with the title bar "GTD - Firefox". The address bar displays the URL "http://localhost:3000/contexts/list". The main content area is titled "GTD" and shows a breadcrumb navigation path: "» Contexts » Projects » Actions » Resources". Below this, the heading "Contexts" is displayed. A link "New Context" is visible. A table lists contexts with columns "Name" and "Actions". The table contains two rows: one for "@Home" with actions "Edit or Delete", and one for "@Work" with actions "Edit or Delete". At the bottom of the page is a "Done" button with a checked checkbox.

Se agora, com um usuário não administrativo, tentarmos acessar um *controller* que deveria ser somente para o outro perfil, teremos um redirecionamento. Isso acontece porque a ação que faz a negação de acesso também está sujeita ao filtro. Resolvemos isso com a seguinte mudança em nosso *controller* primário:

```
class ApplicationController < ActionController::Base

  before_filter :authenticate

  skip_before_filter :authenticate_administration, :only => [:access_denied]

  helper_method :session_user

  protected

    def session_user
      @session_user ||= User.find(:first, :conditions => ['id = ?', session[:user]])
    end

    def authenticate
      unless session[:user]
        session[:return_to] = request.request_uri
        redirect_to :controller => "login", :action => "login"
        return false
      end
      return true
    end

    def authenticate_administration
      unless session_user && session_user.admin?
        redirect_to :action => "access_denied"
        return false
      end
      return true
    end
  end
```

Note que os filtros criados foram aplicados em sua ordem de declaração, partindo do filtro presente no *controller* base e que podemos excluir um filtro para determinadas ações mesmo antes de declararmos o uso do mesmo.

Precisamos agora apenas criar a nossa ação *access_denied*. Como essa é uma ação que será compartilhada por todos os *controllers*, também vamos definir no mesmo arquivo:

```
class ApplicationController < ActionController::Base

  before_filter :authenticate

  skip_before_filter :authenticate_administration, :only => [:access_denied]

  helper_method :session_user

  def access_denied
    render :template => "shared/access_denied"
  end

  protected

    def session_user
      @session_user ||= User.find(:first, :conditions => ['id = ?', session[:user]])
    end
```

```

end

def authenticate
  unless session[:user]
    session[:return_to] = request.request_uri
    redirect_to :controller => "login", :action => "login"
    return false
  end
  return true
end

def authenticate_administration
  unless session[:user] && session[:user].admin?
    redirect_to :action => "access_denied"
    return false
  end
  return true
end

end

```

Note que estamos usando um *template* compartilhado também. Precisamos disso porque, como mencionado anteriormente, o Rails acha *views* pelo *controller*. Usando o método acima, garantimos que o mesmo arquivo sempre será chamado.

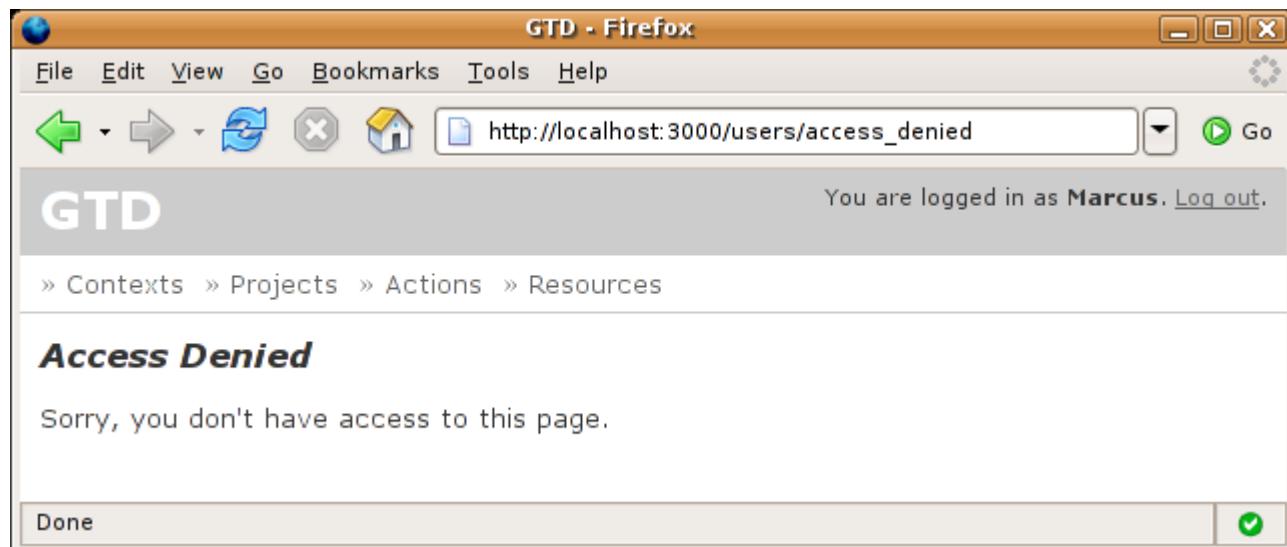
Definimos agora o arquivo, que ficará o diretório `app/views/shared`, com o nome `access_denied.rhtml`, para o seguinte:

```

<h1>Access Denied</h1>
<p>Sorry, you don't have access to this page.</p>

```

O resultado, com um usuário que não tenha acesso, é o seguinte:



Qualquer página administrativa agora poderia fazer uso do filtro, sem precisar se preocupar em definir a

sua própria ação ou sua própria *view*. Aqui vamos que todos os conceitos do modelo orientado a objetos funcionam no Rails e que o *framework*, ao contrário de muitos, não prejudica o desenvolvedor com definições à parte da linguagem.

Nossa aplicação agora contém um mecanismo básico de autenticação e podemos continuar na implementação de mais coisas interessantes na mesma.

UPLOADS

Uma das coisas que podemos fazer de maneira relativamente simples com o Rails é gerenciar o *upload* e *download* de arquivos.

Em uma aplicação como a nossa, cada ação poderia, por exemplo, estar associada a um ou mais arquivos. Faz sentido pensar também que um arquivo pode estar associado a mais de uma ação: por exemplo, você pode precisar de um documento para elaborar uma proposta e também para conversar com o seu cliente sobre a mesma. Isso implicaria em um gerenciador de arquivos ou recursos em nossa aplicação e um modo de associá-los com nossas ações.

A primeira coisa que temos que fazer agora, então, é criar uma maneira de enviar arquivos para a nossa aplicação. Para isto, vamos começar com o nosso modelo de dados:

```
ronaldo@minerva:~/tmp/gtd$ script/generate model resource
exists  app/models/
exists  test/unit/
exists  test/fixtures/
create  app/models/resource.rb
create  test/unit/resource_test.rb
create  test/fixtures/resources.yml
exists  db/migrate
create  db/migrate/007_create_resources.rb
```

E editar a migração para gerar a nossa tabela correspondente:

```
class CreateResources < ActiveRecord::Migration
  def self.up
    create_table :resources do |t|
      t.column :name, :string
      t.column :filename, :string
    end
  end

  def self.down
    drop_table :resources
  end
end
```

No começo, precisamos somente de armazenar o nome do recurso e o nome do seu arquivo. Se precisarmos de mais alguma coisa, sempre podemos voltar e corrigir o nosso modelo com uma nova migração

Precisamos agora de um *controller* para fazer o cadastro de nossos recursos:

```
ronaldo@minerva:~/tmp/gtd$ script/generate controller resources index list edit delete
exists  app/controllers/
exists  app/helpers/
create  app/views/resources
exists  test/functional/
create  app/controllers/resources_controller.rb
create  test/functional/resources_controller_test.rb
create  app/helpers/resources_helper.rb
create  app/views/resources/index.rhtml
create  app/views/resources/list.rhtml
create  app/views/resources/edit.rhtml
create  app/views/resources/delete.rhtml
```

Uma modificação rápida em nosso modelo garante a nossa validação de dados:

```
class Resource < ActiveRecord::Base
  validates_presence_of :name
  validates_presence_of :filename
end
```

Podemos editar os arquivos para chegar ao mesmo padrão que nossos demais, modificando apenas o que precisamos. Primeiro, adicionamos mais uma área em nosso *menu* principal:

```
<html>
<head>
  <title>GTD</title>
  <%= stylesheet_link_tag "default" %>
  <%= stylesheet_link_tag "scaffold" %>
</head>

<body>
  <h1 id="header">GTD</h1>
  <% if session_user %>
  <p id="login-information">
    You are logged in as <strong><%= session_user.name %></strong>.
    <%= link_to "Log out", :controller => "login", :action => "logout" %>.
  </p>
  <% end %>

  <ul id="menu">
    <li><%= link_to_unless_current "&raquo; Contexts", :controller => "contexts", :action => "list" %></li>
    <li><%= link_to_unless_current "&raquo; Projects", :controller => "projects", :action => "list" %></li>
    <li><%= link_to_unless_current "&raquo; Actions", :controller => "actions", :action => "list" %></li>
    <li><%= link_to_unless_current "&raquo; Resources", :controller => "resources", :action => "list" %></li>
    <% if session_user && session_user.admin? %>
      <li><%= link_to_unless_current "&raquo; Users", :controller => "users", :action => "list" %></li>
    <% end %>
  </ul>

  <div id="contents"><%= yield %></div>
```

```
</body>  
</html>
```

Agora podemos criar o *controller* a exemplo dos outros:

```
class ResourcesController < ApplicationController  
  
  def index  
    list  
    render :action => "list"  
  end  
  
  def list  
    @resource_pages, @resources = paginate :resources, :per_page => 5, :order => "name"  
  end  
  
  def edit  
    if params[:id]  
      @resource = Resource.find(params[:id])  
    else  
      @resource = Resource.new  
    end  
    if request.post?  
      @resource.attributes = params[:resource]  
      if @resource.save  
        flash[:notice] = "The resource was successfully saved"  
        redirect_to :action => "list"  
      end  
    end  
  end  
  
  def delete  
    Resource.find(params[:id]).destroy  
    flash[:notice] = "The resource was successfully deleted"  
    redirect_to :action => "list"  
  end  
end
```

E também uma página para listagem:

```
<h1>Resources</h1>  
  
<% if flash[:notice] %>  
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>  
<% end %>  
  
<p><%= link_to "New Resource", :action => "edit" %></p>  
  
<table border="1" cellpadding="5" cellspacing="1">  
  <tr>  
    <th>Name</th>  
    <th>File</th>  
    <th>Actions</th>  
  </tr>  
  <% @resources.each do |resource| %>  
  <tr>  
    <td><%= resource.name %></td>  
    <td><%= resource.filename %></td>  
    <td>  
      <%= link_to "Edit", :action => "edit", :id => resource %> or  
      <%= link_to "Delete", { :action => "delete", :id => resource }, :confirm => "Are you sure?" %>  
    </td>  
  </tr>  
<% end %>
```

```

</tr>
<% end %>
</table>

<p><%= pagination_links(@resource_pages) %></p>

```

Por fim, uma *view* básica para edição:

```

<h1><% if @resource.new_record? %>New<% else %>Editing<% end %> Resource</h1>

<%= error_messages_for "resource" %>

<%= start_form_tag({:action => "edit", :id => @resource}, {:multipart => true}) %>

<p>
  <label for="resource_name">Name:</label><br>
  <%= text_field "resource", "name" %>
</p>

<p>
  <label for="resource_filename">File:</label><br>
  <%= file_field "resource", "filename" %>
</p>

<p>
  <%= submit_tag "Save" %> or <%= link_to "Cancel", :action => "list"%>
</p>

<%= end_form_tag %>

```

Note duas coisas: primeiro, o uso do método `file_field` para gerar um campo de *upload*. Segundo, o uso extendido do método `start_form_tag`, passando um parâmetro adicional de opções. Sem esse parâmetro, `multipart`, que gera o atributo `enctype` do formulário, o formulário não fará *uploads*, enviando somente o nome do arquivo ao invés de seus dados. Essa mudança é vista abaixo, no HTML gerado:

```

<form action="/resources/edit/" enctype="multipart/form-data" method="post">

```

Temos agora a seguinte tela:

GTD - Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:3000/resources/edit

GTD

You are logged in as **Ronaldo**. [Log out.](#)

» Contexts » Projects » Actions » Resources

New Resource

Name:

File: [Browse...](#)

[Save](#) or [Cancel](#)

Done

Se salvarmos um registro, notaremos algumas coisas estranhas: primeiro, nenhum arquivo é enviado para qualquer diretório no servidor; segundo, a validação não funciona; e terceiro, o campo no banco de dados com informações estranhas, como podemos ver na tela abaixo:

GTD - Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:3000/resources/list

GTD

You are logged in as **Ronaldo**. [Log out.](#)

» Contexts » Projects » Actions » **Resources**

Resources

[New Resource](#)

Name	File	Actions
Babel	--- !ruby/object:StringIO {}	Edit or Delete

Done

Isso acontece porque, no caso de um *upload*, um objeto especial é enviado para o *controller* e é responsabilidade do mesmo processar esses objetos. O que vemos acima é uma representação textual simplificada desse objeto.

O objeto `StringIO` representa uma coleção arbitrária de dados binários, acessíveis como um *string*. Podemos usar os métodos desse objeto para salvar o nosso arquivo em disco e fazer o que mais precisamos.

A maneira mais fácil de fazer isso é modificar o modo como o modelo processa os dados, mantendo o uso do atributo transparente para o usuário. Para isto, vamos modificar inicialmente a *view* para gravar em um outro atributo:

```
<h1><% if @resource.new_record? %>New<% else %>Editing<% end %> Resource</h1>

<%= error_messages_for "resource" %>

<%= start_form_tag({:action => "edit", :id => @resource}, {:multipart => true}) %>

<p>
  <label for="resource_name">Name:</label><br>
  <%= text_field "resource", "name" %>
</p>

<p>
  <label for="resource_file">File:</label><br>
  <%= file_field "resource", "file" %>
</p>

<p>
  <%= submit_tag "Save" %> or <%= link_to "Cancel", :action => "list" %>
</p>

<%= end_form_tag %>
```

Modificaremos também o modelo:

```
class Resource < ActiveRecord::Base

  attr_protected :filename

  validates_presence_of :name
  validates_presence_of :filename

  def file=(value)
    end

end
```

O código acima protege o atributo `filename` de qualquer atribuição automática e cria um novo atributo somente de escrita chamado `file` que receberá nossos dados. Testando o formulário você verá que a validação voltou a funcionar embora, obviamente, como não estamos atribuindo nada ao atributo `filename`, o registro não será salvo.

O que precisamos agora é verificar se algum dado foi realmente enviado e salvar em disco se for o caso:

```

class Resource < ActiveRecord::Base

attr_protected :filename

validates_presence_of :name
validates_presence_of :filename

def file=(value)
  if value.size > 0
    self.filename = File.basename(value.original_filename)
    filepath = "#{RAILS_ROOT}/public/uploads/#{self.filename}"
    File.open(filepath, "wb") do |f|
      f.write(value.read)
    end
  end
end

end

```

Antes de executar o código acima, lembre-se de criar um diretório chamado `uploads`, debaixo do diretório `public`, com permissões de escrita e leitura apropriadas para o seu sistema.

No código, verificamos inicialmente se algum dado foi enviado. Depois, recuperamos o nome completo do arquivo original e recolhemos somente o nome do arquivo em si, sem seu caminho, colocando-o no atributo `filename`, e efetivamente validando o mesmo.

Depois disso, geramos um caminho local para o arquivo baseado na variável `RAILS_ROOT`, que é interna ao Rails. Essa variável aponta para a raiz de sua aplicação onde todos os diretórios estão localizados. Vamos aqui também o nosso primeiro exemplo de interpolação: variáveis aplicadas diretamente em um texto, usando o marcador `#{}.` Finalmente, escrevemos os dados recebidos em nossa variável para o arquivo em disco.

Note que escrever um arquivo em disco é uma tarefa extremamente simples em Rails, usando o método `open` da classe `File`, que recebe um bloco como parâmetro, onde as operações são efetuadas. Em nosso caso aqui, escrevemos os dados lidos da variável do tipo `StringIO` recebida. Os métodos `write` e `read`, caso sejam usados da forma acima, sem parâmetros indicativos de tamanho, gravam ou lêem tudo o que receberam. Com o uso do bloco, o arquivo é automaticamente fechado depois de sua utilização.

O código acima tem uma falha fundamental no sentido que sobrescreve arquivos já enviados se o nome for o mesmo de outro registro. Obviamente, sobrescrever os dados arbitrariamente não é uma coisa que queremos. Existem várias formas de resolver isso e veremos uma delas em breve.

Se você testar a aplicação agora, verá que pode fazer o `upload` de qualquer arquivo. Precisamos agora melhorar somente a visualização do que temos. A primeira coisa é colocar um `link` para o arquivo na listagem:

```

<h1>Resources</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New Resource", :action => "edit" %></p>



| Name                 | File                                                                       | Actions                                                                                                                                                        |
|----------------------|----------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <%= resource.name %> | <td><%= link_to resource.filename, "/uploads/#{resource.filename}" %></td> | <%= link_to "Edit", :action => "edit", :id => resource %> or<br><%= link_to "Delete", { :action => "delete", :id => resource }, :confirm => "Are you sure?" %> |



<p><%= pagination_links(@resource_pages) %></p>

```

O resultado vemos abaixo:

Name	File	Actions
Assassin	a1.odt	Edit or Delete
Babel	babel.doc	Edit or Delete

Podemos também melhorar um pouco a nossa página de edição:

```

<h1><% if @resource.new_record? %>New<% else %>Editing<% end %> Resource</h1>

```

```

<%= error_messages_for "resource" %>

<%= start_form_tag({:action => "edit", :id => @resource}, {:multipart => true}) %>



<label for="resource_name">Name:</label><br>
    <%= text_field "resource", "name" %>



<% unless @resource.new_record? %>


The current file is <%= link_to @resource.filename, "/uploads/#{@resource.filename}" %>.
    Choose a new file below to replace it.


<% end %>



<label for="resource_file">File:</label><br>
    <%= file_field "resource", "file" %>



<%= submit_tag "Save" %> or <%= link_to "Cancel", :action => "list" %>



<%= end_form_tag %>

```

Com o seguinte resultado:

The screenshot shows a Firefox browser window with the title 'GTD • Firefox'. The address bar displays the URL <http://localhost:3000/resources/edit/3>. The page content is titled 'GTD' and shows the user is logged in as 'Ronaldo'. The main content area is titled 'Editing Resource' and contains a form for editing a resource. The 'Name:' field is filled with 'Babel'. Below it, a message says 'The current file is babel.doc. Choose a new file below to replace it.' There is a 'File:' input field with a 'Browse...' button. At the bottom of the form are 'Save' and 'Cancel' buttons, and a 'Done' button with a checkmark icon.

Você provavelmente notou que repetimos duas vezes o código que gera a URL do arquivo na aplicação. Seria interessante extraímos isso em um método que possa ser reusado. Fazemos isso em nosso modelo:

```

class Resource < ActiveRecord::Base

  attr_protected :filename

  validates_presence_of :name
  validates_presence_of :filename

  def file=(value)
    if value.size > 0
      self.filename = File.basename(value.original_filename)
      filepath = "#{RAILS_ROOT}/public/uploads/#{self.filename}"
      File.open(filepath, "wb") do |f|
        f.write(value.read)
      end
    end
  end

  def url
    "/uploads/" + self.filename unless self.filename.nil?
  end

end

```

Agora podemos usar esse atributo em nossas *views*. Por exemplo, alterando a *view* de listagem de dados:

```

<h1>Resources</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New Resource", :action => "edit" %></p>

<table border="1" cellpadding="5" cellspacing="1">
  <tr>
    <th>Name</th>
    <th>File</th>
    <th>Actions</th>
  </tr>
  <% @resources.each do |resource| %>
  <tr>
    <td><%= resource.name %></td>
    <td><%= link_to resource.filename, resource.url %></td>
    <td>
      <%= link_to "Edit", :action => "edit", :id => resource %> or
      <%= link_to "Delete", { :action => "delete", :id => resource }, :confirm => "Are you sure?" %>
    </td>
  </tr>
  <% end %>
</table>

<p><%= pagination_links(@resource_pages) %></p>

```

O mesmo pode ser feito na *view* de edição:

```

<h1><% if @resource.new_record? %>New<% else %>Editing<% end %> Resource</h1>

<%= error_messages_for "resource" %>

<%= start_form_tag({:action => "edit", :id => @resource}, {:multipart => true}) %>

<p>

```

```

<label for="resource_name">Name:</label><br>
<%= text_field "resource", "name" %>
</p>

<% unless @resource.new_record? %>
<p>
  The current file is <%= link_to @resource.filename, @resource.url %>.
  Choose a new file below to replace it.
</p>
<% end %>

<p>
  <label for="resource_file">File:</label><br>
  <%= file_field "resource", "file" %>
</p>

<p>
  <%= submit_tag "Save" %> or <%= link_to "Cancel", :action => "list" %>
</p>

<%= end_form_tag %>

```

Um outro detalhe a tratar é o que fazer quando um arquivo é trocado. Nesse caso precisamos remover o arquivo antigo antes de salvar o novo. Precisamos também remover o arquivo do disco quando um registro é excluído do banco. Para isso, nossas soluções estão todas no modelo de dados. Trocar o arquivo é tão simples quanto o código abaixo:

```

class Resource < ActiveRecord::Base

attr_protected :filename

validates_presence_of :name
validates_presence_of :filename

def file=(value)
  if value.size > 0
    if self.filename
      filepath = "#{RAILS_ROOT}/public/uploads/#{self.filename}"
      if File.exists?(filepath)
        File.delete(filepath)
      end
    end
    self.filename = File.basename(value.original_filename)
    filepath = "#{RAILS_ROOT}/public/uploads/#{self.filename}"
    File.open(filepath, "wb") do |f|
      f.write(value.read)
    end
  end
end

def url
  "/uploads/" + self.filename unless self.filename.nil?
end

end

```

E para excluirmos o arquivo em disco se o registro for removido, vamos utilizar uma outra característica do Rails.

CALLBACKS

Callbacks são muito parecidos com os filtros que vimos anteriormente, mas se aplicam a modelos de dados. São métodos que podem ser usados para interceptar os momentos em que um objeto é persistido para o banco de dados e executar alguma ação nesse momento.

Se você consultar a documentação do *ActiveRecord*, verá que o ciclo de persistência de um objeto que nunca tenha sido salvo antes segue as seguintes chamadas:

- (-) save
- (-) valid?
- (1) before_validation
- (2) before_validation_on_create
- (-) validate
- (-) validate_on_create
- (3) after_validation
- (4) after_validation_on_create
- (5) before_save
- (6) before_create
- (-) create
- (7) after_create
- (8) after_save

No exemplo acima, os métodos numerados são *callbacks*, ou seja, locais onde você pode interferir no processamento das demais chamadas. Você pode interferir antes e depois da validação, antes e depois do salvamento e ser tão específico como quiser, dizendo se quer fazer isso somente na primeira vez que o objeto for salvo ou se para todas as demais vezes em que o objeto for persistido.

O mesmo é válido para todas as outras operações de persistência que podem se efetuadas em um objeto. Vamos, então, nos aproveitar disso para interceptar o momento em que o objeto é removido: mais precisamente, logo depois de que o mesmo tenha sido removido. O nosso modelo ficará assim:

```
class Resource < ActiveRecord::Base

attr_protected :filename

validates_presence_of :name
validates_presence_of :filename

after_destroy :remove_file

def file=(value)
  if value.size > 0
    if self.filename
      self.remove_file
    end
    self.filename = File.basename(value.original_filename)
  end
end
```

```

    File.open(self.filepath, "wb") do |f|
      f.write(value.read)
    end
  end
end

def url
  "/uploads/" + self.filename unless self.filename.nil?
end

def filepath
  "#{RAILS_ROOT}/public/uploads/#{self.filename}"
end

def remove_file
  if File.exists?(self.filepath)
    File.delete(self.filepath)
  end
end

end

```

Isso resolve o nosso problema. Note que efetuamos um pequeno refatoramento para eliminar a necessidade de repetir o código.

Como mencionamos acima, a maneira de fazer *uploads* em Rails é bem prática e não necessita de muitos cuidados especiais. O único problema do código acima é que teríamos que ser um pouco mais criteriosos e verificar se algum arquivo não está sendo sobreescrito e tratar isso. Mas, não precisamos nos dar a esse trabalho. Existe uma solução muito mais fácil para casos de *uploads* simples com esse.

PLUGINS

Plugins são uma forma de aumentar a funcionalidade do Rails sem mexer em qualquer de suas bibliotecas básicas. O Rails possui um mecanismo de plugins tão poderoso, graças à flexibilidade da linguagem Ruby, que qualquer coisa pode ser modificada em sua estrutura.

Existem *plugins*, para criar sistemas de autenticação de vários tipos (como inclusive já mencionamos); *plugins* para acrescentar novos tipos de dados às classes, como enumerados, polígonos e números complexos entre outros; *plugins* para criar novas formas de validação; *plugins* para formatar datas, texto, gerar HTML, XML, e dezenas de outras tarefas que você pode analisar por si próprio visitando o repositório localizando em <http://plugins.radrails.org/>.

Um *plugin* em particular pode ajudar a nossa tarefa com *uploads*. Chamado de `file_column`, ele não é só capaz de realizar todo o processo de *upload* mostrado acima, como também permitir um série de outras manipulações no arquivo. Entre essas manipulações incluem-se a possibilidade de geração de *thumbnails* e redimensionamentos de arquivos de imagens. Neste último caso, a *gem* Rmagick (uma biblioteca gráfica para o Ruby) é requerida.

Instalar um *plugin* em Rails é bem fácil. Existem duas formas: baixar o arquivo do mesmo e descompatá-lo na pasta `vendor/plugins`, localizado na aplicação, ou usar um comando próprio.

O comando direto para isso possui características mais avançadas que incluem a possibilidade de integração com o sistema de controle de versão Subversion, e geralmente é a opção mais indicada. Baixar o arquivo na mão pode ser vantajoso para projetos em que mudanças de versão são rigorosamente controladas e você quer exercer decisões manual inclusive sobre os *plugins*, preferindo realizar suas próprias atualizações e correções.

Seja qual for a sua escolha, a sintaxe do comando é bem simples. Você pode ver a ajuda do mesmo rodando o comando abaixo:

```
ronaldo@minerva:~/tmp/gtd$ script/plugin
```

Como você pode ver, ele é bem similar ao comando *gem*.

Para baixar o *plugin* que precisamos aqui, *file_column*, você pode usar o comando seguinte:

```
ronaldo@minerva:~/tmp/gtd$ script/plugin install \
http://opensvn.csie.org/rails_file_column/plugins/file_column/trunk
```

No nosso caso, eu já tenho o *plugin* instalado em outro projeto e vou simplesmente copiá-lo para o diretório *vendor/plugins*, gerando a seguinte estrutura:

```
ronaldo@minerva:~/tmp/gtd/vendor/plugins/file_column$ ls -l
total 28
-rw-r--r-- 1 ronaldo ronaldo 3246 2006-09-22 16:39 CHANGELOG
-rw-r--r-- 1 ronaldo ronaldo 399 2006-09-22 16:39 init.rb
drwxr-xr-x 2 ronaldo ronaldo 4096 2006-09-22 16:40 lib
-rw-r--r-- 1 ronaldo ronaldo 782 2006-09-22 16:39 Rakefile
-rw-r--r-- 1 ronaldo ronaldo 1856 2006-09-22 16:39 README
drwxr-xr-x 4 ronaldo ronaldo 4096 2006-09-22 16:40 test
-rw-r--r-- 1 ronaldo ronaldo 243 2006-09-22 16:39 TODO
```

Você pode explorar o diretório para visualizar a organização interna do *plugin* e entender como o Rails consegue indentificá-lo e incorporá-lo. Lembre-se agora de reiniciar o servidor para que o *plugin* seja carregado pelo Rails.

Agora que temos esse *plugin*, podemos usá-lo em nosso código. Nossa modelo será substituído por:

```
class Resource < ActiveRecord::Base
  validates_presence_of :name
  validates_presence_of :filename
  file_column :filename
end
```

A declaração `file_column` vem diretamente do *plugin* e gera uma série de métodos e atributos que podemos utilizar transparentemente em nossa aplicação.

Se você testou a validação com a nossa versão anterior do modelo de dados e das *views* terá percebido que quando o nome do objeto não é informado, causando uma validação, qualquer arquivo informado será perdido, exigindo que o usuário o informe mais uma vez. No caso do nosso *plugin*, ele toma conta disso armazenando o arquivo temporariamente para que o usuário não tenha que conviver com um comportamento anômalo e nem exigindo que o desenvolvedor se preocupe com detalhes como esses.

Vamos modificar agora nossa *view* de edição para:

```
<h1><% if @resource.new_record? %>New<% else %>Editing<% end %> Resource</h1>

<%= error_messages_for "resource" %>

<%= start_form_tag({:action => "edit", :id => @resource}, {:multipart => true}) %>

<p>
  <label for="resource_name">Name:</label><br>
  <%= text_field "resource", "name" %>
</p>

<% unless @resource.new_record? %>
<p>
  The current file is <%= link_to @resource.base_filename, url_for_file_column("resource",
"filename") %>.
  Choose a new file below to replace it.
</p>
<% end %>

<p>
  <label for="resource_filename">File:</label><br>
  <%= file_column_field "resource", "filename" %>
</p>

<p>
  <%= submit_tag "Save" %> or <%= link_to "Cancel", :action => "list"%>
</p>

<%= end_form_tag %>
```

E nossa *view* de listagem para:

```
<h1>Resources</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New Resource", :action => "edit" %></p>

<table border="1" cellpadding="5" cellspacing="1">
  <tr>
    <th>Name</th>
    <th>File</th>
    <th>Actions</th>
```

```

</tr>
<% @resources.each do |resource| %>
<tr>
  <td><%= resource.name %></td>
  <td>
    <%= link_to resource.base_filename, url_for_file_column(resource, "filename") %>
  </td>
  <td>
    <%= link_to "Edit", :action => "edit", :id => resource %> or
    <%= link_to "Delete", { :action => "delete", :id => resource }, :confirm => "Are you sure?" %>
  </td>
</tr>
<% end %>
</table>

<p><%= pagination_links(@resource_pages) %></p>

```

Pelo fato do *plugin* não ter nenhum método para retornar simplesmente o nome do arquivo, introduzimos a mudança acima na *view* e criamos o método `base_filename`, que você pode ver abaixo:

```

class Resource < ActiveRecord::Base

  validates_presence_of :name
  validates_presence_of :filename

  file_column :filename

  def base_filename
    File.basename(self.filename)
  end

end

```

Como o código do *plugin* é aberto, poderíamos ter corrigido essas deficiências diretamente no mesmo e provavelmente ela será sanada em alguma versão futura. Aliás, com os recursos dinâmicos do Ruby, poderíamos ter introduzido o método diretamente nas classes envolvidas sem necessidades de tocar o código.

Mesmo com esse pequeno problema, porém, podemos ver que o uso de *plugins* pode reduzir substancialmente o trabalho do programador e ajudar o código a ficar mais limpo e mais fácil de ser mantido. Considere, por exemplo, uma aplicação com dezenas de necessidades de *upload*. Ao invés de termos que lidar atributo por atributo com o problema, podemos simplesmente deixar que o *plugin* faça isso por nós.

Veja que os arquivos estão em um diretório sob `public`, cujo nome é a classe de dados, com um diretório para cada atributo controlado pelo *plugin* e, dentro desse diretório, um ouro diretório por objeto. Veja um exemplo na listagem abaixo:

```

ronaldo@minerva:~/tmp/gtd/public/resource/filename/7$ ls
babel.doc

```

Muito simples e intuitivo. Caso você queira controlar os arquivos com maior rigor, autenticando o acesso aos

mesmos, por exemplo, você poderia mudar o diretório onde os mesmos são armazenados usando o código abaixo:

```
file_column :filename, :store_dir => "/uploads/resources/filename/"
```

Seria difícil conseguir algo mais simples do que isso.

Agora que temos os nossos recursos, podemos demonstrar mais uma característica do Rails, que são as associações entre duas classes intermediadas por outra classe, mencionadas mais acima no texto: associações `has_and_belongs_to_many`.

[HAS AND BELONGS TO MANY](#)

Associações `has_and_belongs_to_many` são o resultado de relacionamentos N para N entre duas tabelas em um banco de dados. O Rails é perfeitamente capaz de representar esse tipo de associação ou relacionamento de uma maneira muito similar à feita com `has_many`.

Essas associações são diferentes das associações `has_many` usando *through* no sentido que existe uma tabela real entre os dois modelos enquanto que nesta última o modelo intermediário é usado somente para colapsar os dados em uma agregação.

Como temos uma tabela intermediária representando a relação, o Rails também segue uma convenção. A tabela deve se nomeada como a união entre os nomes dos dois modelos que lhe dão origem, em ordem alfabética. No nosso caso, a tabela que relaciona as ações aos recursos em disco será chamada `actions_resources`. Note o plural nos nomes.

Um detalhe interessante é que caso essa tabela contenha outros campos, eles serão refletidos automaticamente na associação sem necessidade de código adicional. E, mais do que isso, não é necessário gerar qualquer modelo.

Vamos partir então para a migração necessária:

```
ronaldo@minerva:~/tmp/gtd$ script/generate migration create_actions_resources
      create db/migrate
      create db/migrate/008_create_actions_resources.rb
```

E a editamos:

```
class CreateActionsResources < ActiveRecord::Migration
  def self.up
    create_table :actions_resources, :id => false do |t|
      t.column :action_id, :integer
      t.column :resource_id, :integer
```

```

    end
end

def self.down
  drop_table :actions_resources
end
end

```

Como não precisamos de uma coluna automática *id*, informamos à migração que isso pode ser ignorado.

Agora, podemos modificados nossos modelos. Modificaremos ambos porque podemos utilizar a relação nas duas direções.

Começamos com o modelo para as ações:

```

class Action < ActiveRecord::Base

attr_protected :created_at, :completed_at

def done=(value)
  if ActiveRecord::ConnectionAdapters::Column.value_to_boolean(value)
    self.completed_at = DateTime.now
  else
    self.completed_at = nil
  end
  write_attribute("done", value)
end

belongs_to :context
belongs_to :project

validates_presence_of :description

validates_presence_of :context_id
validates_presence_of :project_id

has_and_belongs_to_many :resources

end

```

E depois passamos para o modelo que representa recursos:

```

class Resource < ActiveRecord::Base

validates_presence_of :name
validates_presence_of :filename

file_column :filename

def base_filename
  File.basename(self.filename)
end

has_and_belongs_to_many :actions

end

```

Usando o console agora, podemos manipular essas associações para entendemos um pouco melhor como

elas funcionam:

```
ronaldo@minerva:~/tmp/gtd$ script/console
Loading development environment.

>> action = Action.find(1)
=> #<Action:0xb778a978 @attributes={"context_id"=>"1", "completed_at"=>nil, "project_id"=>"3",
"done"=>"0", "id"=>"1", "description"=>"Buy a grammar to help me with my English.",
"created_at"=>"2006-09-22 09:19:37"}>

>> action.resources
=> []
```

Como podemos ver, nossa associação já está funcionando. Testando um pouco mais:

```
>> action.resources << Resource.find(7)
=> [#<Resource:0xb773cef8 @attributes={"name"=>"Babel", "id"=>"7", "filename"=>"babel.doc"}>

>> action.resources << Resource.find(8)
=> [#<Resource:0xb773cef8 @attributes={"name"=>"Babel", "id"=>"7", "filename"=>"babel.doc"}>,
#<Resource:0xb772fd98 @attributes={"name"=>"First Floor", "id"=>"8", "filename"=>"1st-floor.png"}>

>> action.resources.size
=> 2
```

Como temos a associação do outro lado, podemos ver que a relação é simples:

```
>> resource = Resource.find(7)
=> #<Resource:0xb77e83cc @attributes={"name"=>"Babel", "id"=>"7", "filename"=>"babel.doc"}>

>> resource.actions
=> [#<Action:0xb77a9cf8 @attributes={"context_id"=>"1", "completed_at"=>nil, "project_id"=>"3",
"done"=>"0", "resource_id"=>"7", "id"=>"1", "description"=>"Buy a grammar to help me with my
English.", "action_id"=>"1", "created_at"=>"2006-09-22 09:19:37"}>]
```

No caso de associações com atributos na relação, existe um método especial que pode ser usado para adicionar os valores dos mesmos. Suponha um relação entre uma tabela representando um pedido e uma tabela representando um item do pedido que se relaciona a um produto. Obviamente, no item do pedido teríamos a quantidade pedida e o preço unitário do momento em que o pedido foi gerado. Poderíamos ter um código similar ao abaixo para carregar um novo item em nosso pedido:

```
order = Order.find(params[:order_id])
product = Product.find(params[:product_id])

order.items.push_with_attributes(product, :price => product.price, :quantity => 1)
```

Com o código abaixo, a tabela `items` no banco de dados receberia um novo registro com os campos `order_id` e `product_id` já configurados para os valores próprios dos objetos relacionados e com os campos `price` and `quantity` carregados com os valores passados explicitamente para o método.

Uma vez feito isso, os atributos poderiam ser editados com maior simplicidade:

```
order.items[0].quantity = 2  
order.save
```

Como você pode ver, o relacionamento é refletido intuitivamente no modelo de dados sem necessidade de qualquer esforço adicional. Para toda a funcionalidade gerada, precisamos apenas de duas linhas de código.

A documentação descreve uma série de outros métodos e parâmetros que podem ser usados para lidar com as situações mais inusitadas que você tiver. Se quisermos, por exemplo, ordenar nossas associações, podemos mudar um de nossos modelos para:

```
class Action < ActiveRecord::Base  
  
attr_protected :created_at, :completed_at  
  
def done=(value)  
  if ActiveRecord::ConnectionAdapters::Column.value_to_boolean(value)  
    self.completed_at = DateTime.now  
  else  
    self.completed_at = nil  
  end  
  write_attribute("done", value)  
end  
  
belongs_to :context  
belongs_to :project  
  
validates_presence_of :description  
  
validates_presence_of :context_id  
validates_presence_of :project_id  
  
has_and_belongs_to_many :resources, :order => "name"  
end
```

Para aproveitarmos essas mudanças em nosso modelo, vamos trabalhar mais uma vez com o *controller* que gerencia o cadastro de ações. O nosso objetivo agora é permitir que o usuário associe recursos de arquivos a uma ação qualquer. Para isto, teremos um cadastro interno que nos permitirá múltiplas associações de cada vez.

Começamos, desta vez, modificando a ação de listagem das ações:

```
<h1>Actions</h1>  
  
<% if flash[:notice] %>  
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>  
<% end %>  
  
<p><%= link_to "New Action", :action => "edit" %></p>  
  
<table border="1" cellpadding="5" cellspacing="1">  
  <tr>  
    <th>Description</th>  
    <th>Completed</th>
```

```

<th>When</th>
<th>Project</th>
<th>Context</th>
<th>Actions</th>
</tr>
<% @actions.each do |action| %>
<tr>
  <td><%= action.description %></td>
  <td><%= yes_or_no?(action.done?) %></td>
  <td><%= (action.done?) ? action.completed_at.strftime("%m/%d/%y") : "-" %></td>
  <td><%= action.project.name %></td>
  <td><%= action.context.name %></td>
  <td nowrap>
    <%= link_to "Edit", :action => "edit", :id => action %> or<br>
    <%= link_to "Delete", { :action => "delete", :id => action }, :confirm => "Are you sure?" %>
or<br>
    <%= link_to "Edit Resources", :action => "resources", :id => action %>
  </td>
</tr>
<% end %>
</table>

<p><%= pagination_links(@action_pages) %></p>

```

E em seguida o *controller* das ações:

```

class ActionsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @action_pages, @actions = paginate :actions, :include => [:project, :context], :per_page => 5,
:order => "actions.description"
  end

  def edit
    @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    @projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    if params[:id]
      @item = Action.find(params[:id])
    else
      @item = Action.new
    end
    if request.post?
      @item.attributes = params[:item]
      if @item.save
        flash[:notice] = "The action was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def resources
    @action = Action.find(params[:id])
  end

  def delete
    Action.find(params[:id]).destroy
    flash[:notice] = "The action was successfully deleted"
    redirect_to :action => "list"
  end
end

```

Com essa variável, já podemos criar uma *view* que inicialmente exibe os recursos associados com um ação:

```
<h1>Action Resources</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p>
  <strong>Action Description:</strong><br>
  <%= @action.description %>
</p>

<h3>Resources</h3>

<% if @action.resources.size > 0 %>

  <table border="1" cellpadding="5" cellspacing="1">
    <tr>
      <th>Name</th>
      <th>File</th>
      <th>Actions</th>
    </tr>
    <% @action.resources.each do |resource| %>
    <tr>
      <td><%= resource.name %></td>
      <td>
        <%= link_to resource.base_filename, url_for_file_column(resource, "filename") %>
      </td>
      <td>
        <%= link_to "Delete", { :action => "delete_resource", :id => @action, :resource_id => resource }, :confirm => "Are you sure?" %>
      </td>
    </tr>
    <% end %>
  </table>

<% else %>

  <p><em>No resource was added to this action yet.</em></p>

<% end %>
```

O resultado é:

The screenshot shows a Firefox browser window with the title 'GTD - Firefox'. The address bar displays the URL 'http://localhost:3000/actions/resources/2'. The main content area of the browser shows a web page titled 'Action Resources'. The page includes a 'Action Description:' section with the text 'Get permission from the powers that be to plant a tree in the lounge.' Below this is a 'Resources' section with the message 'No resource was added to this action yet.' At the bottom of the page is a 'Done' button.

Precisamos de uma forma de buscar as ações que queremos associar. Podemos usar um formulário simples para isso:

```
<h1>Action Resources</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p>
  <strong>Action Description:</strong><br>
  <%= @action.description %>
</p>

<h3>Resources</h3>

<% if @action.resources.size > 0 %>

  <table border="1" cellpadding="5" cellspacing="1">
    <tr>
      <th>Name</th>
      <th>File</th>
      <th>Actions</th>
    </tr>
    <% @action.resources.each do |resource| %>
    <tr>
      <td><%= resource.name %></td>
      <td>
        <% @resource = resource %>
        <%= link_to resource.base_filename, url_for_file_column("resource", "filename") %>
      </td>
      <td>
        <%= link_to "Delete", { :action => "delete_resource", :id => @action, :resource_id => resource }, :confirm => "Are you sure?" %>
      </td>
    </tr>
    <% end %>
```

```

</table>

<% else %>



<em>No resource was added to this action yet.</em></p>



<% end %>

<%= start_form_tag :action => "resources" %>



### Search for Resources to Add



<label for="keywords">Keywords:</label><br>
    <%= text_field_tag "keywords", params[:keywords] %>



<input type="submit" value="Search" />



<%= end_form_tag %>

<% if @resources.size > 0 %>

<%= start_form_tag :action => "add_resources", :id => @action %>



<strong>Results</strong>




<% @resources.each do |resource| %>
- <input checked="" type="checkbox" value="<%= resource.id %>" />
        <%= resource.name %><br>
        <span><%= link_to resource.base_filename, url_for_file_column(resource, "filename") %></span>

<% end %>



<input type="submit" value="Add" />



<%= end_form_tag %>

<% else %>



<em>This search returned to matches.</em></p>



<% end %>

```

Agora, mudamos o nosso *controller* para:

```

class ActionsController < ApplicationController

def index
    list
    render :action => "list"
end

def list
    @action_pages, @actions = paginate :actions, :include => [:project, :context], :per_page => 5,
:order => "actions.description"
end

def edit
    @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }

```

```

@projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
if params[:id]
  @item = Action.find(params[:id])
else
  @item = Action.new
end
if request.post?
  @item.attributes = params[:item]
  if @item.save
    flash[:notice] = "The action was successfully saved"
    redirect_to :action => "list"
  end
end
end

def resources
  @action = Action.find(params[:id])
  @resources = []
  if request.post? && !params[:keywords].blank?
    @resources = Resource.find(:all, :conditions => ['name like ? or filename like ?',
"%#{params[:keywords]}%", "%#{params[:keywords]}%"])
  end
end

def delete
  Action.find(params[:id]).destroy
  flash[:notice] = "The action was successfully deleted"
  redirect_to :action => "list"
end
end

```

E adicionamos o seguinte fragmento à nossa *stylesheet* padrão, em public/stylesheets/default.css:

```

ul.search-results
{
  list-style: none;
  padding: 0;
  margin: 0;
}

ul.search-results li
{
  margin-bottom: 5px;
}

ul.search-results li span
{
  font-size: smaller;
}

```

Com essas mudanças, temos o seguinte resultado:

You are logged in as **Ronaldo**. [Log out.](#)

» Contexts » Projects » Actions » Resources

Action Resources

Action Description:
Buy a grammar to help me with my English.

Resources

No resource was added to this action yet.

Search for Resources to Add

Keywords:

Results

- First Floor
[1st-floor.png](#)
- Second Floor
[2nd-floor.png](#)

Precisamos agora realmente adicionar os recursos escolhidos à ação. Para isso, criamos a ação que já especificamos na *view*:

```
class ActionsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @action_pages, @actions = paginate :actions, :include => [:project, :context], :per_page => 5,
    :order => "actions.description"
  end

```

```

def edit
  @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
  @projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
  if params[:id]
    @item = Action.find(params[:id])
  else
    @item = Action.new
  end
  if request.post?
    @item.attributes = params[:item]
    if @item.save
      flash[:notice] = "The action was successfully saved"
      redirect_to :action => "list"
    end
  end
end

def resources
  @action = Action.find(params[:id])
  @resources = []
  if request.post? && !params[:keywords].blank?
    @resources = Resource.find(:all, :conditions => ['name like ? or filename like ?',
    "%#{params[:keywords]}%", "%#{params[:keywords]}%"])
  end
end

def add_resources
  @action = Action.find(params[:id])
  existing_resource_ids = @action.resources.collect { |resource| resource.id }
  new_resource_ids = params[:resources].reject { |id| existing_resource_ids.include?(id.to_i) }
  @action.resource_ids << new_resource_ids
  redirect_to :action => "resources", :id => @action.id
end

def delete
  Action.find(params[:id]).destroy
  flash[:notice] = "The action was successfully deleted"
  redirect_to :action => "list"
end
end

```

Essa ação verifica os recursos existentes, encontra os ainda não adicionados àquela ação, os adiciona e, por fim, redireciona para a listagem mais uma vez. Verificar recursos já adicionados previne a inclusão de múltiplos recursos na tabela intermediária.

Finalmente, precisamos da ação que remove um registro associado, que é bem simples:

```

class ActionsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @action_pages, @actions = paginate :actions, :include => [:project, :context], :per_page => 5,
    :order => "actions.description"
  end

  def edit
    @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    @projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    if params[:id]

```

```

    @item = Action.find(params[:id])
  else
    @item = Action.new
  end
  if request.post?
    @item.attributes = params[:item]
    if @item.save
      flash[:notice] = "The action was successfully saved"
      redirect_to :action => "list"
    end
  end
end

def resources
  @action = Action.find(params[:id])
  @resources = []
  if request.post? && !params[:keywords].blank?
    @resources = Resource.find(:all, :conditions => ['name like ? or filename like ?',
"%#{params[:keywords]}%", "%#{params[:keywords]}%"])
  end
end

def add_resources
  @action = Action.find(params[:id])
  existing_resource_ids = @action.resources.collect { |resource| resource.id }
  new_resource_ids = params[:resources].reject { |id| existing_resource_ids.include?(id.to_i) }
  @action.resources << Resource.find(new_resource_ids)
  redirect_to :action => "resources", :id => @action.id
end

def delete_resource
  @action = Action.find(params[:id])
  @action.resources.delete(Resource.find(params[:resource_id]))
  redirect_to :action => "resources", :id => @action.id
end

def delete
  Action.find(params[:id]).destroy
  flash[:notice] = "The action was successfully deleted"
  redirect_to :action => "list"
end
end

```

Como é possível perceber, o uso de associações, seja qual forem, é muito simples no Rails. Fica como exercício para o leitor limitar a busca com base nos registros já existentes de modo que, para conveniência do usuário, somente registros ainda não adicionados apareçam.

ESCOPOS

Uma outra característica interessante do Rails, introduzida na versão 1.1, é o conceito de escopos para operações no banco de dados. O conceito é muito similar às declarações *with* presentes em várias linguagens de programação, que temporariamente aumentam o nível de escopo sintático de uma ou mais variáveis, facilitando o uso das mesmas.

O uso de escopos, a exemplo das declarações *with*, pode simplificar bastante o uso do *ActiveRecord*, mas deve ser empregado com cuidado por possibilitar a introdução de *bugs* sutis no código que podem ser bem difíceis de depurar.

Para demonstrar o uso de escopos, vamos fazer algumas alterações em nossa aplicação, expandido a

possibilidade de uso do sistema por múltiplos usuários. Até o momento, qualquer usuário, mesmo um administrador, podia acessar todas as ações, contextos e projetos gerados. Obviamente, isso não é o que queremos. Com a ajuda de escopos, podemos resolver isso rapidamente.

Primeiramente, vamos alterar o nosso modelo de dados, isolando cada objeto para conter também a informação do usuário que o criou. Geramos uma migração:

```
ronaldo@minerva:~/tmp/gtd$ script/generate migration add_user_to_objects
exists db/migrate
create db/migrate/009_add_user_to_objects.rb
```

E a editamos:

```
class AddUserToObjects < ActiveRecord::Migration
  def self.up
    add_column :contexts, :user_id, :integer
    add_column :projects, :user_id, :integer
    add_column :actions, :user_id, :integer
    add_column :resources, :user_id, :integer
    Context.reset_column_information
    Project.reset_column_information
    Action.reset_column_information
    Resource.reset_column_information
    user = User.find_by_name("ronaldo")
    Context.update_all "user_id = #{user.id}"
    Project.update_all "user_id = #{user.id}"
    Action.update_all "user_id = #{user.id}"
    Resource.update_all "user_id = #{user.id}"
  end

  def self.down
    remove_column :contexts, :user_id
    remove_column :projects, :user_id
    remove_column :actions, :user_id
    remove_column :resources, :user_id
  end
end
```

No caso de uma migração como essa, precisamos também atualizar o nosso banco. E isso o que faz essa migração. Primeiro, os modelos são recarregados. Depois disso, encontramos o primeiro usuário que criamos. E por fim, utilizamos o método `update_all` para fazer atualizações em massa em nossos registros.

Agora, vamos experimentar com o *controller* responsável pelos contextos, utilizando escopos. Se não estivéssemos usando escopos e quiséssemos, por exemplo, limitar a nossa ação *list* a somente registros de um usuário, teríamos algo assim:

```
class ContextsController < ApplicationController
  def index
    list
    render :action => "list"
  end

  def list
```

```

    @context_pages, @contexts = paginate :contexts, :conditions => ["user_id = ?", session[:user]],
:per_page => 5, :order => "name"
end

def edit
  if params[:id]
    @context = Context.find(params[:id])
  else
    @context = Context.new
  end
  if request.post?
    @context.attributes = params[:context]
    if @context.save
      flash[:notice] = "The context was successfully saved"
      redirect_to :action => "list"
    end
  end
end

def delete
  Context.find(params[:id]).destroy
  flash[:notice] = "The context was successfully deleted"
  redirect_to :action => "list"
end
end

```

É uma maneira perfeitamente válida de fazer isso. Com escopos, teríamos:

```

class ContextsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    Context.with_scope(:find => { :conditions => ["user_id = ?", session[:user]] }) do
      @context_pages, @contexts = paginate :contexts, :per_page => 5, :order => "name"
    end
  end

  def edit
    if params[:id]
      @context = Context.find(params[:id])
    else
      @context = Context.new
    end
    if request.post?
      @context.attributes = params[:context]
      if @context.save
        flash[:notice] = "The context was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def delete
    Context.find(params[:id]).destroy
    flash[:notice] = "The context was successfully deleted"
    redirect_to :action => "list"
  end
end

```

O método `with_scope` recebe um série de declarações que definem condições e atributos a serem aplicados. A princípio pode parecer mais esforço, mas o importante a entender na chamada acima é que todas as

chamadas ao banco dentro do bloco usando o método `find` e seus derivados estariam sujeitas ao escopo criado para o mesmo, recebendo automaticamente as condições especificadas. No que tange ao método `find`, basicamente qualquer de seus parâmetros pode ser especificado, como `offset` e `limit`, por exemplo.

Alterando a ação `edit` teríamos algo assim:

```
class ContextsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    Context.with_scope(:find => { :conditions => ["user_id = ?", session[:user]] }) do
      @context_pages, @contexts = paginate :contexts, :per_page => 5, :order => "name"
    end
  end

  def edit
    Context.with_scope(:find => { :conditions => ["user_id = ?", session[:user]] }, :create =>
    { :user_id => session[:user] }) do
      if params[:id]
        @context = Context.find(params[:id])
      else
        @context = Context.new
      end
      if request.post?
        @context.attributes = params[:context]
        if @context.save
          flash[:notice] = "The context was successfully saved"
          redirect_to :action => "list"
        end
      end
    end
  end

  def delete
    Context.find(params[:id]).destroy
    flash[:notice] = "The context was successfully deleted"
    redirect_to :action => "list"
  end
end
```

O escopo aqui não só limita a busca, como também limita a criação, informando automaticamente o atributo `user_id` quando necessário.

Para impedir que o atributo fosse automaticamente informado em um formulário, precisaríamos mudar o nosso modelo de dados:

```
class Context < ActiveRecord::Base

  attr_protected :user_id

  validates_presence_of :name
  validates_uniqueness_of :name

  validates_presence_of :user_id
```

```
has_many :actions  
end
```

Com o escopo que estabelecemos, vemos que se um usuário tentar acessar diretamente um contexto criado por outro usuário, teremos um erro:

The screenshot shows a Firefox browser window with the title "Action Controller: Exception caught - Firefox". The address bar displays the URL "http://localhost:3000/contexts/edit/1". The main content area shows the following error message:
ActiveRecord::RecordNotFound in ContextsController#edit
Couldn't find Context with ID=1
RAILS_ROOT: script/../config/..
[Application Trace](#) | [Framework Trace](#) | [Full Trace](#)
/usr/lib/ruby/gems/1.8/gems/activerecord-1.14.4/lib/active_record/base.rb:955:in `find'_
/usr/lib/ruby/gems/1.8/gems/activerecord-1.14.4/lib/active_record/base.rb:941:in `find'_
/usr/lib/ruby/gems/1.8/gems/activerecord-1.14.4/lib/active_record/base.rb:382:in `find'_
app/controllers/contexts_controller.rb:17:in `edit'_
app/controllers/contexts_controller.rb:15:in `edit'_

Request
Parameters: {"id"=>"1"}
[Show session dump](#)

Response
Headers: {"cookie"=>[], "Cache-Control"=>"no-cache"}

E vemos que, em nosso arquivo log/development.log, o SQL gerado continha a condição informada no escopo:

```

Processing ContextsController#edit (for 127.0.0.1 at 2006-09-24 12:40:45) [GET]
Session ID: 1c381dc70993b556066506987d635218
Parameters: {"action"=>"edit", "id"=>"1", "controller"=>"contexts"}
Context Load (0.004735)    SELECT * FROM contexts WHERE (user_id = 2) AND (contexts.id = '1') LIMIT
1

```

Há um pequeno problema no código acima, porém. Se você testou a criação de um registro, verá que o escopo não se aplicou ao método `save`, e, consequentemente, o registro foi salvo sem um atributo `user_id`.

A explicação para isso poderia ser dupla: poderíamos dizer que o problema está no nosso uso de `save`, tanto para criação quanto para atualização, ou poderíamos também dizer que há um problema no Rails, já que, se examinarmos o código do mesmo, veremos que o escopo não é aplicado em todas situações em que um objeto é criado. E, de fato, existem alguns problemas com o método para os quais soluções já foram propostas na base de código de desenvolvimento do Rails e pode ser estarão em suas versões futuras.

A solução, no momento, é fazer uma modificação na ação `edit`:

```

class ContextsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    Context.with_scope(:find => { :conditions => ["user_id = ?", session[:user]] }) do
      @context_pages, @contexts = paginate :contexts, :per_page => 5, :order => "name"
    end
  end

  def edit
    Context.with_scope(:find => { :conditions => ["user_id = ?", session[:user]] }, :create =>
{ :user_id => session[:user] }) do
      if request.get?
        if params[:id]
          @context = Context.find(params[:id])
        else
          @context = Context.new
        end
      elsif request.post?
        @context = params[:id] ?
          Context.update(params[:id], params[:context]) :
          Context.create(params[:context])
        if @context.valid?
          flash[:notice] = "The context was successfully saved"
          redirect_to :action => "list"
        end
      end
    end
  end

  def delete
    Context.find(params[:id]).destroy
    flash[:notice] = "The context was successfully deleted"
    redirect_to :action => "list"
  end
end

```

Obviamente, essa é uma solução menos do que ideal por exigir código extra, mas é o melhor que podemos

fazer no momento.

No código acima, notamos que seria interessante ter o mesmo escopo aplicado a todas as ações, para prevenir qualquer forma de acesso sem parâmetros. Uma solução simples é criar um método que nos retorne os escopos e aplicar isso a todos as ações de maneira mais limpa:

```
class ContextsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    Context.with_scope(needed_scopes) do
      @context_pages, @contexts = paginate :contexts, :per_page => 5, :order => "name"
    end
  end

  def edit
    Context.with_scope(needed_scopes) do
      if request.get?
        if params[:id]
          @context = Context.find(params[:id])
        else
          @context = Context.new
        end
      elsif request.post?
        @context = params[:id] ?
          Context.update(params[:id], params[:context]) :
          Context.create(params[:context])
        if @context.valid?
          flash[:notice] = "The context was successfully saved"
          redirect_to :action => "list"
        end
      end
    end
  end

  def delete
    Context.with_scope(needed_scopes) do
      Context.find(params[:id]).destroy
      flash[:notice] = "The context was successfully deleted"
      redirect_to :action => "list"
    end
  end

  protected

    def needed_scopes
      @needed_scopes ||= {
        :find => { :conditions => ["user_id = ?", session[:user]] },
        :create => { :user_id => session[:user] }
      }
    end
  end
```

Essa solução é bem mais legível do que ficar repetindo escopos em cada uma das ações, mas podemos melhorá-la ainda mais, indo direto à fonte. Em especial, queremos uma solução que possa ser aplicada a qualquer controller que precisarmos. Como nossa aplicação é bem simples, podemos fazer isso através de um filtro especialmente criado para isso.

Um única ressalva aqui é que estamos entrando em um território no Rails onde não há uma API estabelecida, e o nosso código é sujeito a mudanças de implementação que podem quebrá-lo eventualmente. Mas, no momento, podemos aprender algo com o que vamos fazer.

FILTROS AROUND

Como vamos precisar de uma implementação que seja acessível a todos *controllers*, e de maneira geral a toda aplicação, precisamos colocá-la em algum lugar de nossa aplicação de modo que ela seja carregada automaticamente. Podemos fazer isso usando o diretório `extras`, que é feito especialmente para isso.

Primeiro precisamos habilitar esse diretório em nosso arquivo `config/environment.rb`, que é onde as configurações para a aplicação estão definidas, e depois incluir a chamada ao arquivo que precisamos:

```
# Be sure to restart your web server when you modify this file.

# Uncomment below to force Rails into production mode when
# you don't control web/app server and can't set it the proper way
# ENV['RAILS_ENV'] ||= 'production'

# Specifies gem version of Rails to use when vendor/rails is not present
RAILS_GEM_VERSION = '1.1.6'

# Bootstrap the Rails environment, frameworks, and default configuration
require File.join(File.dirname(__FILE__), 'boot')

Rails::Initializer.run do |config|
  # Settings in config/environments/* take precedence those specified here

  # Skip frameworks you're not going to use (only works if using vendor/rails)
  # config.frameworks -= [ :action_web_service, :action_mailer ]

  # Add additional load paths for your own custom dirs
  config.load_paths += %W( #{RAILS_ROOT}/extras )

  # Force all environments to use the same logger level
  # (by default production uses :info, the others :debug)
  # config.log_level = :debug

  # Use the database for sessions instead of the file system
  # (create the session table with 'rake db:sessions:create')
  # config.action_controller.session_store = :active_record_store

  # Use SQL instead of Active Record's schema dumper when creating the test database.
  # This is necessary if your schema can't be completely dumped by the schema dumper,
  # like if you have constraints or database-specific column types
  # config.active_record.schema_format = :sql

  # Activate observers that should always be running
  # config.active_record.observers = :cacher, :garbage_collector

  # Make Active Record use UTC-base instead of local time
  # config.active_record.default_timezone = :utc

  # See Rails::Configuration for more options
end

# Add new inflection rules using the following format
# (all these examples are active by default):
# Inflector.inflections do |inflect|
#   inflect.plural /^(ox)$/i, '\1\en'
```

```

# inflect.singular /^(ox)en/i, '\1'
# inflect.irregular 'person', 'people'
# inflect.uncountable %w( fish sheep )
# end

# Include your application configuration below
require "user_filter.rb"

```

O arquivo `environment.rb` contém uma série de configurações que podem ser modificadas para otimizar o uso de certas características do Rails. Por exemplo, se você não pretende enviar e-mails de uma aplicação ou usar *web services*, pode impedir que as bibliotecas de apoio sejam carregadas, reduzindo o uso de memória e aumentando a velocidade de sua aplicação.

No arquivo acima, já aproveitamos para incluir o nosso arquivo, que será carregado automaticamente junto com a aplicação. Agora, só precisamos criá-lo (lembre-se de criar o diretório também):

```

class UserFilter

  def initialize(target_class, target_method)
    @target_class = target_class
    @target_method = target_method
  end

  def before(controller)
    filters = controller.send(@target_method)
    @target_class.instance_eval do
      scoped_methods << with_scope(filters) { current_scoped_methods }
    end
  end

  def after(controller)
    @target_class.instance_eval do
      scoped_methods.pop
    end
  end

end

```

A princípio o arquivo acima pode parecer um pouco esotérico, mas é realmente bem simples. `UserFilter` é a primeira classe que estamos criando em nossa aplicação que não se encaixa na estrutura padrão do Rails. Como pensamos em utilizá-la como um filtro do tipo *around*, ou seja, executado antes e depois de uma ação, precisamos definir pelo menos dois métodos: `before` e `after`, que, como os nomes dizem, rodam antes e depois da ação.

No caso acima, precisamos também de um método `initialize` porque vamos passar parâmetros de criação para a classe. Em Ruby, o método `initialize` é chamada quando a classe é instanciada através do método `new`. Quaisquer parâmetros passados para `new` serão passados também para `initialize`.

Para o nosso filtro, precisamos de duas coisas: a classe à qual estaremos aplicando o escopo e qual o escopo a ser aplicado. Como filtros *around* rodam em contextos especiais, o melhor que podemos fazer é passar o nome do método que criamos para definir o filtro e invocá-lo no momento em que precisarmos.

Antes de estudarmos a classe, veja como ficou o *controller* depois da aplicação da mesma:

```
class ContextsController < ApplicationController
  around_filter UserFilter.new(Context, :needed_scopes)

  def index
    list
    render :action => "list"
  end

  def list
    @context_pages, @contexts = paginate :contexts, :per_page => 5, :order => "name"
  end

  def edit
    if request.get?
      if params[:id]
        @context = Context.find(params[:id])
      else
        @context = Context.new
      end
    elsif request.post?
      @context = params[:id] ?
        Context.update(params[:id], params[:context]) :
        Context.create(params[:context])
      if @context.valid?
        flash[:notice] = "The context was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def delete
    Context.find(params[:id]).destroy
    flash[:notice] = "The context was successfully deleted"
    redirect_to :action => "list"
  end

  protected

    def needed_scopes
      @needed_scopes ||= {
        :find => { :conditions => ["user_id = ?", session[:user]] },
        :create => { :user_id => session[:user] }
      }
    end
end
```

Veja que as chamadas diretas a `with_scope` desapareceram, sendo automaticamente aplicadas pelo filtro. Vamos entender agora como a classe funciona.

Dois parâmetros são passados ao filtro em sua criação: a classe que será modificada e um método do próprio *controller* que descreve os filtros a serem aplicados. O método `initialize` da classe `UserFilter` é simples:

```
def initialize(target_class, target_method)
  @target_class = target_class
  @target_method = target_method
end
```

Apenas guardamos as variáveis passadas na instância para serem utilizadas depois. Nada de especial até o momento.

Depois temos o método `before`:

```
def before(controller)
  filters = controller.send(@target_method)
  @target_class.instance_eval do
    scoped_methods << with_scope(filters) { current_scoped_methods }
  end
end
```

Esse método é invocado antes da ação e recebe como parâmetro o `controller` da mesma. A primeira coisa que o método faz aqui é enviar uma mensagem para o `controller` invocando o método que definimos antes como contendo os escopos necessários (no caso aqui, `needed_scopes`). Lembre-se que tudo em Ruby é um objeto. O método `send` pode ser usado em qualquer classe para invocar qualquer método da mesma. A variável `filters` aqui receberá então o resultado da invocação, que são as definições dos escopos necessários.

A segunda linha usa o método `instance_eval` para realizar um pouco de mágica. Esse método recebe um bloco que, ao invés de ser executado no contexto do método atual, é executado no contexto da instância sob o qual foi invocado, no caso aqui, a própria classe `Context`. As classes que são derivadas de `ActiveRecord::Base`, ou seja, todas as classes que definem os nossos modelos de dados, possuem vários métodos e variáveis relacionadas a escopos. Vemos três acima: `with_scope`, `scoped_methods` e `current_scoped_methods`. Já trabalhamos com o primeiro e os dois seguintes descrevem respectivamente uma variável de instância que é `array` com todos os escopos ativos (`find`, `create`, etc.) e um método que retorna o último escopo aninhado já que escopos podem ser empilhados.

A linha de código então faz algo interessante. Primeiro, ela entra em um escopo, usando o próprio método `with_scope`, com os filtros que acabamos de obter do `controller`, e dentro do bloco, invoca o método `current_method_scopes` para retornar o último escopo aninhado que foi adicionado. Esse é o retorno do bloco que então é inserido como mais um item na variável `scoped_methods`, efetivamente ativando um escopo na classe. Você pode estar pensando: mas isso não aplica o escopo duas vezes? Não, pela mágica dos blocos: tão logo o bloco de `with_scope` retorne, o filtro aplicado é removido, seguido pela adição do retorno.

Aqui vai uma coincidência: depois de ter elaborado a versão acima do código, lembrei que existia um *plugin* que fazia justamente isso, de uma maneira bem mais avançada. Você pode encontrar o *plugin* e uma descrição muito mais elaborada e completa da técnica que usamos nesse ponto no endereço seguinte: http://habtm.com/articles/2006/02/22/nested-with_scope. Inclusive, a linha acima é uma adaptação de uma linha do *plugin*, que executa de maneira mais completa o que eu estava fazendo antes.

Finalmente, o método `after` remove o escopo criado no método `before`, removendo-o do array `scoped_methods`, seguindo exatamente o código interno do Rails. O resultado é que, durante o tempo de aplicação do filtro, todas as chamadas ao `controller` estão debaixo dos escopos que precisamos.

Se você acessar a aplicação agora, com usuário diferentes, verá que o `controller` que manipula os contextos é capaz de filtrar perfeitamente os registros de acordo com o usuário autenticado no momento, sem qualquer interferência no código do mesmo.

Em nossa aplicação, queremos que esse filtro seja aplicado a todos os `controllers`. Poderíamos então mover suas chamadas para nosso `controller` primário, que ficaria assim:

```
class ApplicationController < ActionController::Base

  before_filter :authenticate

  skip_before_filter :authenticate_administration, :only => [:access_denied]

  around_filter UserFilter.new(Context, :needed_scopes)
  around_filter UserFilter.new(Project, :needed_scopes)
  around_filter UserFilter.new(Action, :needed_scopes)
  around_filter UserFilter.new(Resource, :needed_scopes)

  helper_method :session_user

  def access_denied
    render :template => "shared/access_denied"
  end

  protected

    def session_user
      @session_user ||= User.find(:first, :conditions => ['id = ?', session[:user]])
    end

    def authenticate
      unless session[:user]
        session[:return_to] = request.request_uri
        redirect_to :controller => "login", :action => "login"
        return false
      end
      return true
    end

    def authenticate_administration
      unless session_user && session_user.admin?
        redirect_to :action => "access_denied"
        return false
      end
      return true
    end

    def needed_scopes
      @needed_scopes ||= {
        :find => { :conditions => ["user_id = ?", session[:user]] },
        :create => { :user_id => session[:user] }
      }
    end
end
```

Depois, removemos o código de nosso `controller` de contextos, que voltaria a:

```

class ContextsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @context_pages, @contexts = paginate :contexts, :per_page => 5, :order => "name"
  end

  def edit
    if request.get?
      if params[:id]
        @context = Context.find(params[:id])
      else
        @context = Context.new
      end
    elsif request.post?
      @context = params[:id] ?
        Context.update(params[:id], params[:context]) :
        Context.create(params[:context])
      if @context.valid?
        flash[:notice] = "The context was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def delete
    Context.find(params[:id]).destroy
    flash[:notice] = "The context was successfully deleted"
    redirect_to :action => "list"
  end
end

```

Como o escopo é aplicado diretamente nas classes de dados, percebemos que nossos *controllers* não precisam de modificações diretas (com exceção, é claro, das mudanças para usar os métodos *create* e *update* ao invés de *save*, dado o problema que percebemos com a aplicação de escopos, que fica aqui como um exercício para o leitor).

Se rodarmos alguns de nossos *controllers*, teremos alguns problemas com as nossas condições, por causa do nome ambíguo de nosso coluna, que aparece em várias classes. Uma forma rápida de resolver isso é mudar o código de nosso filtro para:

```

class UserFilter

  def initialize(target_class, target_method)
    @target_class = target_class
    @target_method = target_method
  end

  def before(controller)
    filters = controller.send(@target_method, @target_class)
    @target_class.instance_eval do
      scoped_methods << with_scope(filters) { current_scoped_methods }
    end
  end

  def after(controller)

```

```

    @target_class.instance_eval do
      scoped_methods.pop
    end
  end

end

```

E o nosso *controller* base para:

```

class ApplicationController < ActionController::Base

  before_filter :authenticate

  skip_before_filter :authenticate_administration, :only => [:access_denied]

  around_filter UserFilter.new(Context, :needed_scopes)
  around_filter UserFilter.new(Project, :needed_scopes)
  around_filter UserFilter.new(Action, :needed_scopes)
  around_filter UserFilter.new(Resource, :needed_scopes)

  helper_method :session_user

  def access_denied
    render :template => "shared/access_denied"
  end

  protected

  def session_user
    @session_user ||= User.find(:first, :conditions => ['id = ?', session[:user]])
  end

  def authenticate
    unless session[:user]
      session[:return_to] = request.request_uri
      redirect_to :controller => "login", :action => "login"
      return false
    end
    return true
  end

  def authenticate_administration
    unless session_user && session_user.admin?
      redirect_to :action => "access_denied"
      return false
    end
    return true
  end

  def needed_scopes(target_class)
    {
      :find => { :conditions => ["#{target_class.table_name}.user_id = ?", session[:user]] },
      :create => { :user_id => session[:user] }
    }
  end
end

```

Usando a própria classe passada como parâmetro, especificamos melhor a condição de filtro.

Isso resolve a maior parte dos nossos problemas e encontramos uma maneira simples de filtrar os registros que precisamos sem ter que espalhar condições por toda a nossa aplicação. Como mencionado anteriormente, escopos deve ser usados com cuidado, mas oferecem soluções extremamente elegantes para

determinados problemas.

XML

Uma das coisas que podemos fazer em Rails é retornar XML para consumo de clientes em qualquer formato que precisamos. Em uma aplicação como a nossa, um exemplo de XML que podemos querer produzir é um *feed RSS*.

Eu imagino que a maioria de nós esteja familiar com esse tipo de tecnologia. Aqueles que não estiverem podem encontrar mais informações em <http://www.rssfificado.com.br/>.

Seria interessante que os usuários da nossa aplicação fosse capazes de acessarem um *feed RSS* com as ações que precisam executar. Poderíamos ter simplesmente uma listagem das ações ainda não realizadas que seria automaticamente atualizada caso um novo registro fosse inserido na base de dados.

Como mencionamos anteriormente, o Rails pode produzir não só saída HTML, mas qualquer outra que seja necessária. No caso aqui, vamos produzir saída XML que é automaticamente suportada pelo mecanismo de templates do Rails. Para começar, vamos gerar um novo *controller*:

```
ronaldo@minerva:~/tmp/gtd$ script/generate controller feed
exists app/controllers/
exists app/helpers/
create app/views/feed
exists test/functional/
create app/controllers/feed_controller.rb
create test/functional/feed_controller_test.rb
create app/helpers/feed_helper.rb
```

Esse *controller* terá somente uma ação que retornará o nosso *feed RSS*:

```
class FeedController < ApplicationController

  def index
    @actions = Action.find :all, :conditions => ["done = 0"],
                          :order => "created_at desc"
    render :layout => false
  end

end
```

Na ação acima, pegamos todas as ações ainda em aberto, ordenadas em data decrescente de criação. Precisamos também renderizar a ação sem layout já que a *view* associada gerará tudo o que precisamos.

Agora, precisamos criar a *view* relacionada com essa ação. Nesse caso, ao invés de gerarmos um arquivo *index.rhtml*, usaremos um arquivo chamado *index.rxml*:

```
xml.instruct!
```

```

xml.rss "version" => "2.0", "xmlns:dc" => "http://purl.org/dc/elements/1.1/" do
  xml.channel do
    xml.title      "Due Actions"
    xml.link       url_for(:only_path => false, :controller => "actions")
    xml.pubDate    CGI.rfc1123_date(@actions.first.created_at) if @actions.any?
    xml.description "Due actions"

    @actions.each do |action|
      xml.item do
        xml.title      action.description
        xml.link       url_for(:only_path => false, :controller => "actions", :action => "edit", :id => action.id)
        xml.pubDate    CGI.rfc1123_date(action.created_at)
        xml.guid       url_for(:only_path => false, :controller => "actions", :action => "edit", :id => action.id)
      end
    end
  end
end

```

A variável `xml` está automaticamente presente e é uma instância da classe `XmlBuilder`, que permite uma aplicação gerar XML quase como se estivesse descrevendo código Ruby.

Por meio de uma aplicação interessante das propriedades da linguagem, a classe `XmlBuilder` intercepta as chamadas a métodos inexistentes e os transforma em declarações automáticas de elementos XML, que podem ser aninhados com o uso de blocos.

A técnica usada é exatamente a mesma gerada para criar os métodos automáticos que temos nas classes de dados, no formato `find_*`, criando o necessário no momento em que a primeira invocação é feita de modo que o desenvolvedor possa utilizar a classe como se todos os atributos do arquivo XML existissem anteriormente.

O resultado é o seguinte:

```
-<rss version="2.0">
  -<channel>
    <title>Due Actions</title>
    <link>http://localhost:3000/actions</link>
    <pubDate>Fri, 22 Sep 2006 12:20:19 GMT</pubDate>
    <description>Feed Description</description>
    -<item>
      <title>Upgrade my word processor.</title>
      <link>http://localhost:3000/actions/edit/4</link>
      <pubDate>Fri, 22 Sep 2006 12:20:19 GMT</pubDate>
      <guid>http://localhost:3000/actions/edit/4</guid>
    </item>
    -<item>
      -<title>
        Sit down and write, every day, at least two hours a day.
      </title>
      <link>http://localhost:3000/actions/edit/3</link>
      <pubDate>Fri, 22 Sep 2006 12:20:09 GMT</pubDate>
      <guid>http://localhost:3000/actions/edit/3</guid>
    </item>
```

Como você pode ver, cada elemento descrito através do objeto `xml` foi gerado, no que não passa de uma view normal utilizando outro formato.

Temos, porém, um problema com o uso desse *feed*. Se o introduzirmos em um leitor RSS, veremos que o mesmo simplesmente falha porque, sem autenticação, tudo o que o que é retornado é um redirecionamento para a página de *login*. Precisamos então de um maneira alternativa de autenticar o nosso *feed*.

AUTENTICAÇÃO HTTP

Uma forma de fazer isso é usar a autenticação já usada por navegadores, cujos tipos principais são Basic, Digest e NTLM. As três são suportadas pela maior parte dos navegadores em existência. A primeira é a forma mais simples, mas mais facilmente interceptável. A terceira é bem segura, mas complexa e proprietária, sendo uma variação do padrão Kerberos usada pela Microsoft. Embora suportada por outros navegadores, não é o que queremos implementar no momento já que a maioria dos agregadores não a

suporta. Ficamos então com a Basic, cujo uso é relativamente simples e que é suportada pelos principais leitores de RSS.

Para fazer podermos usar essa autenticação, precisamos mudar o nosso filtro de *login*.

A autenticação Basic funciona basicamente com o uso de um obscurecimento simples do *login* e senha enviados pelo navegador. Basta recuperarmos esses dados e conseguiremos o que precisamos para identificar o usuário. Vejamos como o nosso *controller* ficaria:

```
class FeedController < ApplicationController

  def index
    @actions = Action.find :all, :conditions => ["done = 0"],
      :order => "created_at desc"
    render :layout => false
  end

  protected

  def authenticate
    login, password = get_basic_data
    user = User.find_by_login_and_password(login, password)
    if user
      session[:user] = user.id
      return true
    else
      headers["Status"] = "Unauthorized"
      headers["WWW-Authenticate"] = "Basic realm=\"GTD\""
      render :text => "Failed to authenticate the provided login and password",
        :status => "401 Unauthorized"
      return false
    end
  end

  def get_basic_data
    login, password = "", ""
    if request.env.has_key?("X-HTTP-AUTHORIZATION")
      data = request.env["X-HTTP-AUTHORIZATION"].to_s.split
    elsif request.env.has_key?("HTTP-AUTHORIZATION")
      data = request.env["HTTP-AUTHORIZATION"].to_s.split
    end
    if data and data[0] == "Basic"
      login, password = Base64.decode64(data[1]).split(":") [0..1]
    end
    return [login, password]
  end
end
```

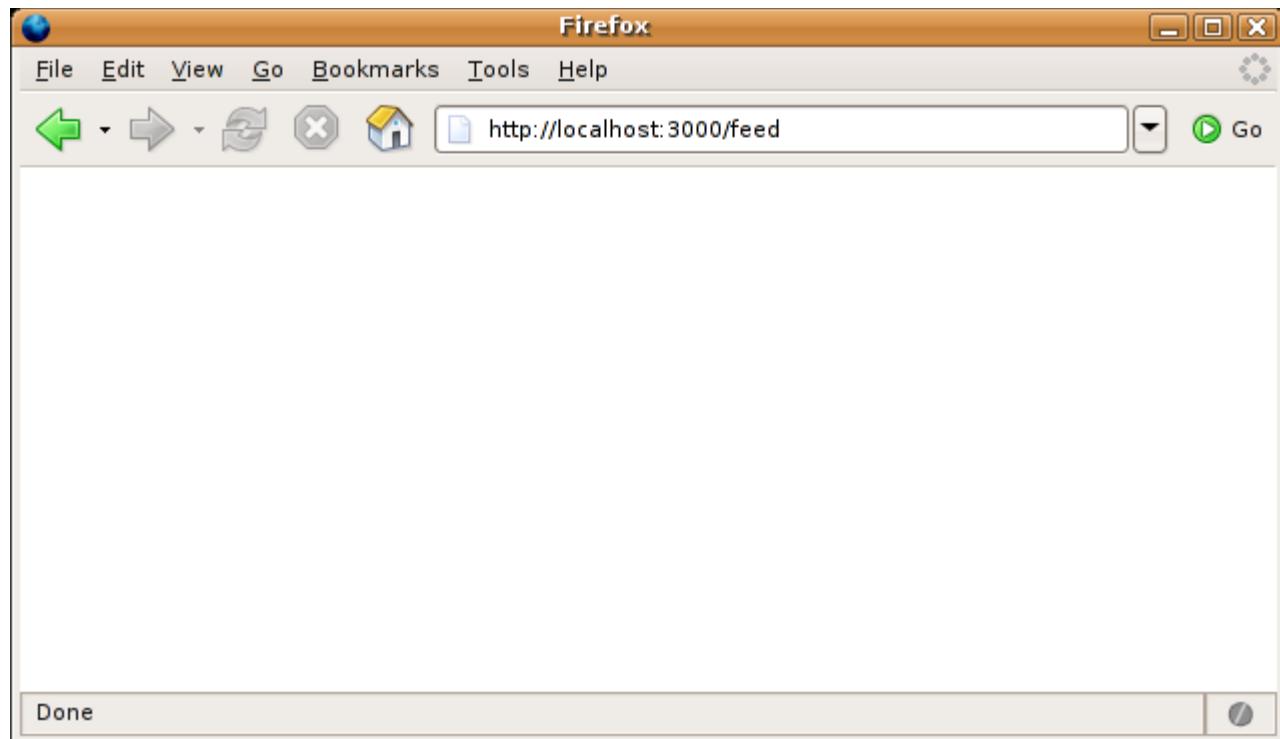
O que fizemos acima foi mudar o filtro que faz a autenticação. De autenticação baseada em uma página de *login*, temos agora autenticação Basic. Como nosso *controller* herda todos os seus métodos do *controller* base, para trocarmos a implementação de um método, basta redefiní-lo. Precisamos fazer isso aqui ao invés de somente criar um outro filtro porque nossos escopos dependem de uma variável de sessão que é inicializada nesse filtro. Se adicionássemos outros, os filtros de escopo falhariam por estarem no *controller* base e rodarem antes de qualquer filtro definido em outro classe.

O novo método *authenticate* é bem simples. Ele verifica se os dados de *login* recebidos correspondem a um

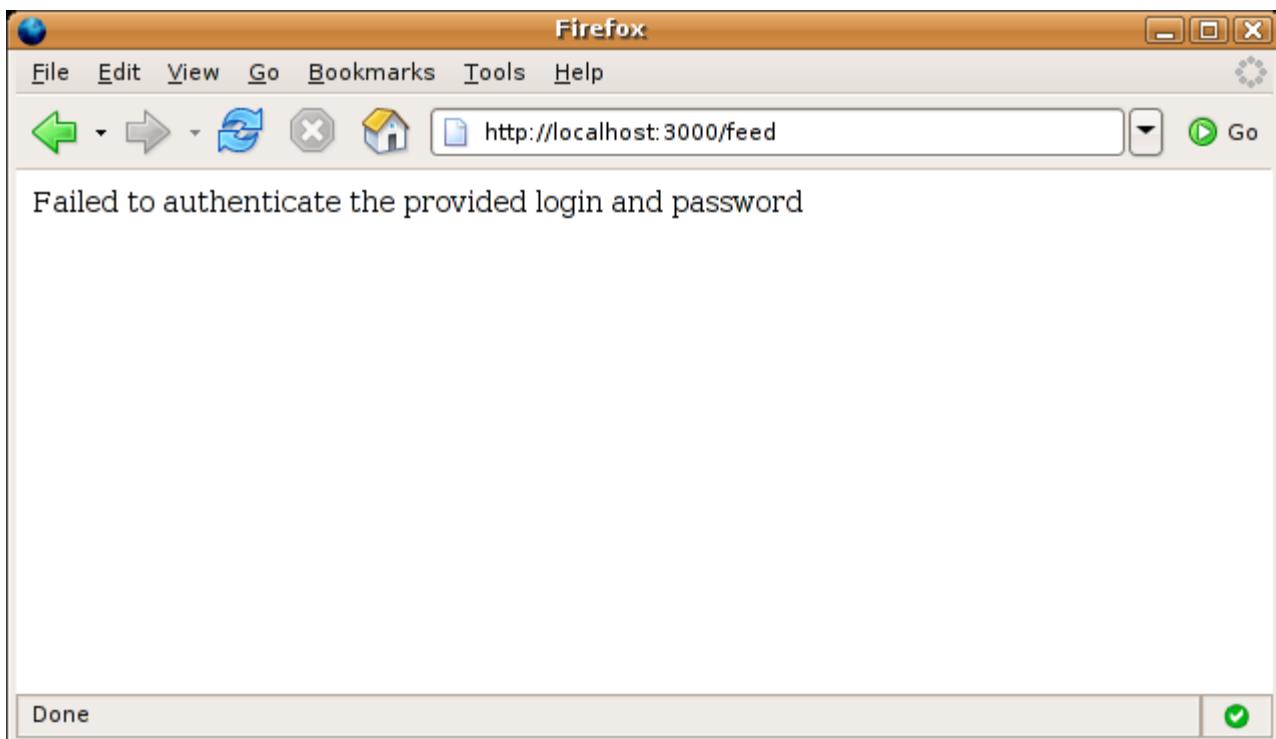
usuário válido. Se sim, os dados são salvos em sessão para uso em nossos filtros anteriores, o valor verdadeiro é retornado. Caso contrário, cabeçalhos indicando que o usuário não foi autorizado são retornados, de acordo com a especificação, junto com o valor false para bloquear qualquer processamento posterior.

O método `get_basic_data`, por sua vez, tenta obter dados de autenticação das variáveis de ambiente geradas pelo servidor em resposta ao navegador. Como esses dados podem estar em dois locais, elas são testados em ordem. Depois disso, se a autenticação for realmente Basic, os dados presentes são decodificados do formato Base64, usado pelo navegador e retornados.

O resultado é o seguinte:



Se o usuário recusa a autenticação, vemos algo assim (no Internet Explorer pode ficar um pouco diferente de acordo com as configurações do mesmo, mas o resultado final é o mesmo):



Precisamos agora somente de uma pequena alteração em nosso *controller* para finalizá-lo:

```
class FeedController < ApplicationController
  skip_before_filter :authenticate
  before_filter :authenticate_basic

  def index
    @actions = Action.find :all, :conditions => ["done = 0"],
    :order => "created_at desc"
    headers["Cache-Control"] = "no-cache"
    headers["Content-Type"] = "application/xml+rss"
    render :layout => false
  end

  protected

  def authenticate_basic
    login, password = get_basic_data
    user = User.find_by_login_and_password(login, password)
    if user
      session[:user] = user.id
      return true
    else
      headers["Status"] = "Unauthorized"
      headers["WWW-Authenticate"] = "Basic realm=\"GTD\""
      render :text => "Failed to authenticate the provided login and password", :status => "401 Unauthorized"
      return false
    end
  end

  def get_basic_data
    login, password = "", ""
    if request.env.has_key?("X-HTTP-AUTHORIZATION")
```

```

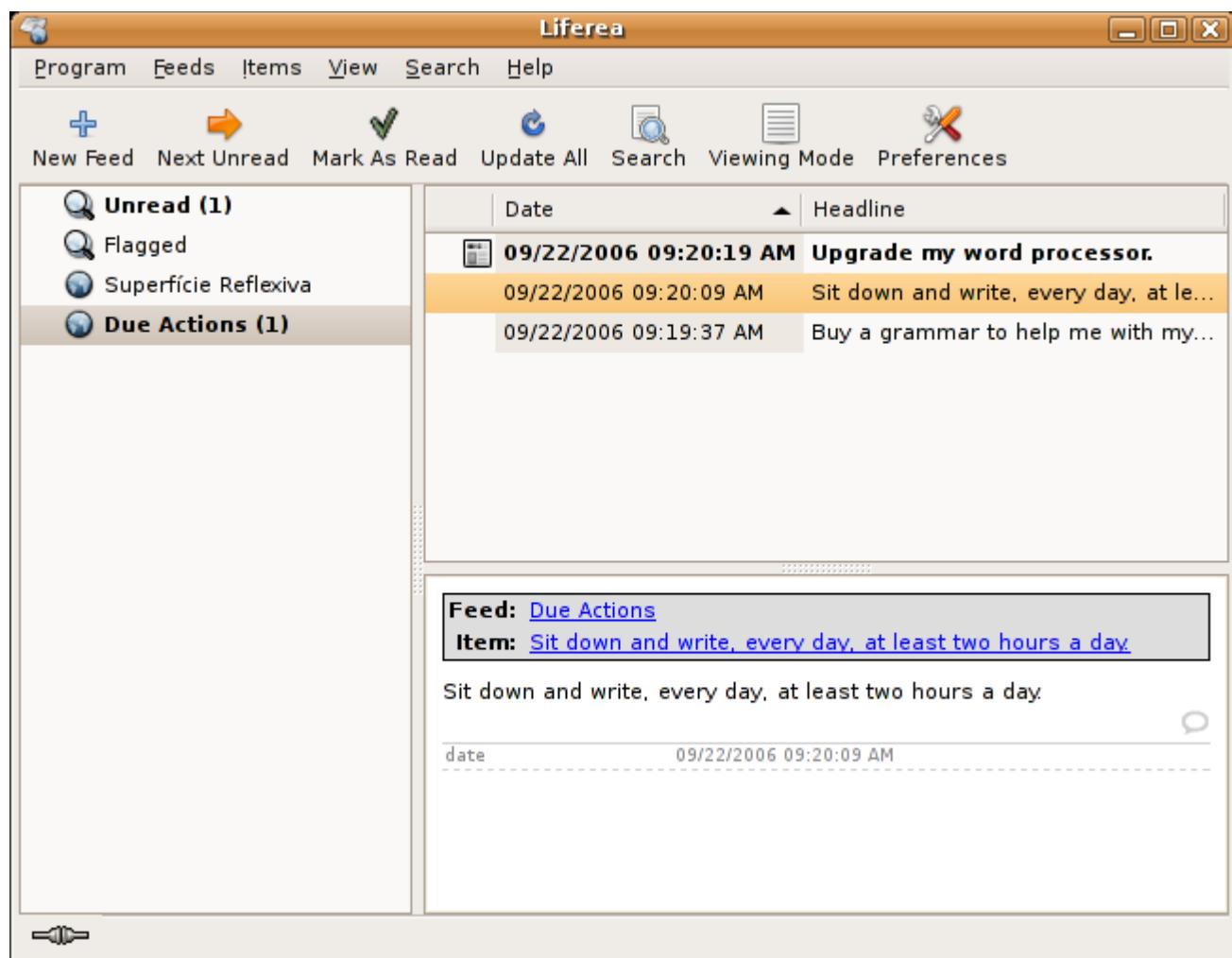
    data = request.env["X-HTTP-AUTHORIZATION"].to_s.split
  elsif request.env.has_key?("HTTP_AUTHORIZATION")
    data = request.env["HTTP_AUTHORIZATION"].to_s.split
  end
  if data and data[0] == "Basic"
    login, password = Base64.decode64(data[1]).split(":") [0..1]
  end
  return [login, password]
end

end

```

A primeira instrução é para que o navegador não use o seu *cache* interno e a segunda indica o tipo de dados que deve ser retornado quando usando RSS.

Agora, podemos ver como o *feed* ficaria em um aplicação RSS própria:



Não são muitas informações, mas você já tem uma idéia do que é possível fazer com o Rails em termos de geração de XML (seja qual for o formato necessário) e autenticação avançada.

Continuando nosso aprendizado, vamos passar a outra parte do Rails.

ENVIANDO E-MAILS

O Rails possui toda uma parte, chamada de *ActionMailer*, responsável pelo envio e recebimento de e-mails. Para fins desse tutorial, vamos somente usar a parte de envio.

Suponhamos que queremos enviar um e-mail para usuário recém-criados, informando-lhes do acesso que eles agora tem ao sistema. Podemos fazer isso facilmente. Primeiramente, é claro, precisamos do e-mail do usuário, que não temos em nosso modelo. Podemos adicionar isso facilmente com uma migração:

```
ronaldo@minerva:~/tmp/gtd$ script/generate migration add_email_to_user
exists db/migrate
create db/migrate/010_add_email_to_user.rb
```

Editando a migração, temos:

```
class AddEmailToUser < ActiveRecord::Migration
  def self.up
    add_column :users, :email, :string
  end

  def self.down
    remove_column :users, :email
  end
end
```

Podemos adicionar validação a esse campos em nosso modelo:

```
class User < ActiveRecord::Base
  validates_presence_of :name
  validates_presence_of :login
  validates_presence_of :password
  validates_presence_of :email

  validates_uniqueness_of :login
end
```

A validação acima somente verifica se o e-mail foi informado, mas não se o mesmo é valido. Existe um *plugin* específico que permite a validação real de e-mails que você pode utilizar nesse caso.

E a listagem de usuário:

```
<h1>Users</h1>
<% if flash[:notice] %>
```

```

<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New User", :action => "edit" %></p>



| Name             | Login             | Administrator                      | E-mail                | Actions                                                                                                                                             |
|------------------|-------------------|------------------------------------|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| <%= user.name %> | <%= user.login %> | <td><%= yes_or_no?(user.admin?) %> | <td><%= user.email %> | <%= link_to "Edit", :action => "edit", :id => user %> or <%= link_to "Delete", { :action => "delete", :id => user }, :confirm => "Are you sure?" %> |


<% @users.each do |user| %>
<% end %>
<p><%= pagination_links(@user_pages) %></p>

```

Finalmente, a edição:

```

<h1><% if @user.new_record? %>New<% else %>Editing<% end %> User</h1>

<%= error_messages_for "user" %>

<%= start_form_tag :action => "edit", :id => @user %>

<p>
  <label for="user_name">Name:</label><br>
  <%= text_field "user", "name" %>
</p>

<p>
  <label for="user_login">Login:</label><br>
  <%= text_field "user", "login" %>
</p>

<p>
  <label for="user_password">Password:</label><br>
  <%= password_field "user", "password" %>
</p>

<p>
  <label for="user_admin">Administrator:</label><br>
  <%= select "user", "admin", [["No", false], ["Yes", true]] %>
</p>

<p>
  <label for="user_email">E-mail:</label><br>
  <%= text_field "user", "email" %>
</p>

<p>
  <%= submit_tag "Save" %> or <%= link_to "Cancel", :action => "list" %>
</p>

<%= end_form_tag %>

```

Agora precisamos de uma forma de enviar um e-mail para novos usuários. Para enviar e-mails, o Rails utiliza *mailers*, que representam tanto o aspecto do *model* como da *view* na estratégia MVC. Geramos um novo com o seguinte comando:

```
ronaldo@minerva:~/tmp/gtd$ script/generate mailer registration_notifier
exists  app/models/
create  app/views/registration_notifier
exists  test/unit/
create  test/fixtures/registration_notifier
create  app/models/registration_notifier.rb
create  test/unit/registration_notifier_test.rb
```

Como você pode ver, esse comando um arquivo em `app/models`, que contém o seguinte:

```
class RegistrationNotifier < ActionMailer::Base
end
```

É nessa classe que criaremos os métodos que geram o e-mail a ser enviado, incluindo os seus dados. Essa classe utiliza uma *view* associado ao método para gerar o conteúdo do e-mail. Começaremos com o método:

```
class RegistrationNotifier < ActionMailer::Base
  def registration_notification(user, session_user)
    recipients "#{user.name} <#{user.email}>"
    from "#{session_user.name} <#{session_user.email}>"
    subject "Registration"
    body "user" => user
  end
end
```

O que esse método faz é gerar os campos e corpo do e-mail. Os dados acima são os básicos. Em ordem: `recipient`, o endereço para o qual o e-mail será enviado; `from`, o endereço de quem enviou; `subject`, o assunto do e-mail; e `body`, o corpo da mensagem, que será gerado a partir de uma *view*, com a variável `user` disponível na mesma através da associação. Vamos aqui mais uma mostra do poder de expressividade do Rails, através do uso de métodos que definem uma linguagem de domínio específica.

Consultando a documentação, você verá que podemos manipular outros cabeçalhos como `cc`, `bcc` e `sent_on`, representando respectivamente os cabeçalhos cópia-carbono, cópia-carbono invisível e data de envio.

Vamos enviar um e-mail simples, mas na documentação você também verá que é possível enviar mensagens HTML e, inclusive, mensagens com partes múltiplas, para uso com anexos e outros dados.

Para gerarmos o corpo da mensagem, precisamos de uma *view* com o mesmo nome do método que criamos acima, ficando o arquivo como `registration_notification.rhtml`, localizado no diretório `app/views/registration_notifier`:

```

Dear <%= @user.name %>:

Welcome to our GTD application. To access it, log in to:

http://localhost:300/home

You login is <%= @user.login %>, and your password is <%= @user.password %>.

Thanks for your interest.

Regards,

The Administrator

```

Como você pode ver, um arquivo simples no mesmo estilo das *views* que já geramos, com a única diferença de ser textual.

Antes de enviar o nosso e-mail pelo *controller*, precisamos de uma modificação na configuração de nossa aplicação: informar o servidor pelo qual os e-mails serão enviados. Isso somente é necessário se você não tiver um servidor local que responda em *localhost*. De qualquer forma, como esse pode ser justamente o seu caso, mostramos aqui a modificação necessária, que acontece no arquivo *config/environments.rb*:

```

# Be sure to restart your web server when you modify this file.

# Uncomment below to force Rails into production mode when
# you don't control web/app server and can't set it the proper way
# ENV['RAILS_ENV'] ||= 'production'

# Specifies gem version of Rails to use when vendor/rails is not present
RAILS_GEM_VERSION = '1.1.6'

# Bootstrap the Rails environment, frameworks, and default configuration
require File.join(File.dirname(__FILE__), 'boot')

Rails::Initializer.run do |config|
  # Settings in config/environments/* take precedence those specified here

  # Skip frameworks you're not going to use (only works if using vendor/rails)
  # config.frameworks -= [ :action_web_service, :action_mailer ]

  # Add additional load paths for your own custom dirs
  config.load_paths += %W( #{RAILS_ROOT}/extras )

  # Force all environments to use the same logger level
  # (by default production uses :info, the others :debug)
  # config.log_level = :debug

  # Use the database for sessions instead of the file system
  # (create the session table with 'rake db:sessions:create')
  # config.action_controller.session_store = :active_record_store

  # Use SQL instead of Active Record's schema dumper when creating the test database.
  # This is necessary if your schema can't be completely dumped by the schema dumper,
  # like if you have constraints or database-specific column types
  # config.active_record.schema_format = :sql

  # Activate observers that should always be running
  # config.active_record.observers = :cacher, :garbage_collector

  # Make Active Record use UTC-base instead of local time
  # config.active_record.default_timezone = :utc

```

```

# See Rails::Configuration for more options
end

# Add new inflection rules using the following format
# (all these examples are active by default):
# Inflector.inflections do |inflect|
#   inflect.plural /^(ox)$/, '\1en'
#   inflect.singular /^(ox)en/, '\1'
#   inflect.irregular 'person', 'people'
#   inflect.uncountable %w( fish sheep )
# end

# Include your application configuration below
require "user_filter.rb"

ActionMailer::Base.server_settings =
{
  :address => "mail.yourdomain.com",
  :domain => "yourdomain.com",
  :user_name => "user@yourdomain.com",
  :password => "*****",
  :authentication => :login
}

```

Edite para os dados necessários e reinicie o servidor para recarregar a nova configuração.

Por fim, precisamos simplesmente modificar o nosso *controller* responsável pelos usuários para o envio da mensagem propriamente dito:

```

class UsersController < ApplicationController

  before_filter :authenticate_administration

  def index
    list
    render :action => "list"
  end

  def list
    @user_pages, @users = paginate :users, :per_page => 5, :order => "name"
  end

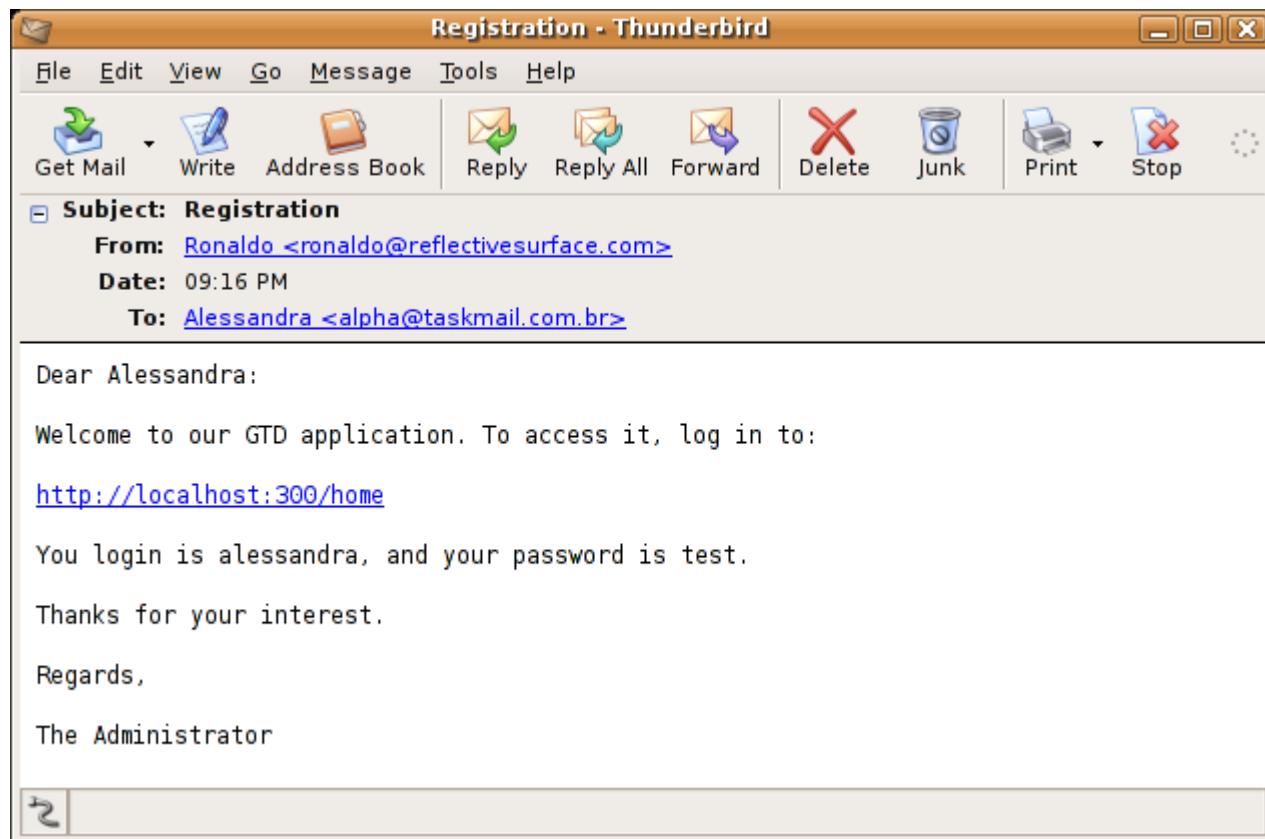
  def edit
    if params[:id]
      @user = User.find(params[:id])
    else
      @user = User.new
    end
    if request.post?
      @user.attributes = params[:user]
      send_email = @user.new_record?
      if @user.save
        if send_email
          RegistrationNotifier.deliver_registration_notification(@user)
        end
        flash[:notice] = "The user was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def delete
    User.find(params[:id]).destroy
    flash[:notice] = "The user was successfully deleted"
  end

```

```
    redirect_to :action => "list"
end
end
```

Após salvar um usuário novo, temos o seguinte resultado em nossa caixa de e-mail:



Obviamente, só queremos enviar um e-mail para novos usuários. Por isso, no código, antes de salvarmos o registro, verificamos se o mesmo é novo. Se é, depois de o salvarmos com sucesso, enviamos o e-mail. Se não, o e-mail é ignorado. Note que você não precisa instanciar qualquer classe. Você precisa apenas adicionar `deliver_` ao método que você gerou e o e-mail será criado e enviado automaticamente.

Como podemos ver, a maneira de enviar e-mail no Rails é bem simples.

DEPURANDO APLICAÇÕES

Uma coisa que o Rails permite e que não vimos até o momento é a depuração de aplicações através de seu console. Essa é uma característica muito simples, mas extremamente útil na identificação rápida de problemas.

Para começarmos, usaremos um novo comando do Rails:

```
ronaldo@minerva:~/tmp/gtd$ script/breakpointer
No connection to breakpoint service at druby://localhost:42531 (DRb::DRbConnError)
Tries to connect will be made every 2 seconds...
```

Esse comando inicial o depurador remoto do Rails, que espera na porta específica por conexões. Com o comando iniciado, podemos introduzir um *breakpoint* em nossa aplicação. Vamos colocar um em um lugar qualquer apenas para demonstração:

```
class UsersController < ApplicationController

  before_filter :authenticate_administration

  def index
    list
    render :action => "list"
  end

  def list
    @user_pages, @users = paginate :users, :per_page => 5, :order => "name"
    breakpoint
  end

  def edit
    if params[:id]
      @user = User.find(params[:id])
    else
      @user = User.new
    end
    if request.post?
      @user.attributes = params[:user]
      send_email = @user.new_record?
      if @user.save
        if send_email
          RegistrationNotifier.deliver_registration_notification(@user, session_user)
        end
        flash[:notice] = "The user was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def delete
    User.find(params[:id]).destroy
    flash[:notice] = "The user was successfully deleted"
    redirect_to :action => "list"
  end
end
```

Voltando ao comando, veremos o seguinte:

```
ronaldo@minerva:~/tmp/gtd$ script/breakpointer
No connection to breakpoint service at druby://localhost:42531 (DRb::DRbConnError)
Tries to connect will be made every 2 seconds...
Executing break point at ./script/../config/../app/controllers/users_controller.rb:12 in `list'
irb(#<UsersController:0xb7735e4c>):001:0>
```

A aplicação agora está parada e pode ser acessada pelo console:

```

irb(#<UsersController:0xb7735e4c>):001:0> @users
=> [#<User:0xb77031a4 @attributes={"name"=>"Alessandra", "admin"=>"0", "id"=>"12",
"password"=>"test", "login"=>"alessandra", "email"=>"alessandra@reflectivesurface.com">,
#<User:0xb770308c @attributes={"name"=>"Marcus", "admin"=>"0", "id"=>"2", "password"=>"test",
"login"=>"marcus", "email"=>"marcus@reflectivesurface.com">], #<User:0xb7702f74
@attributes={"name"=>"Ronaldo", "admin"=>"1", "id"=>"1", "password"=>"test", "login"=>"ronaldo",
"email"=>"ronaldo@reflectivesurface.com">]
irb(#<UsersController:0xb7735e4c>):002:0>

```

Testando mais um pouco:

```

irb(#<UsersController:0xb7735e4c>):011:0> instance_variables
=> ["@response", "@performed_redirect", "@headers", "@user_pages", "@action_name", "@flash",
"@template", "@cookies", "@__bp_file", "@url", "@performed_render", "@before_filter_chain_aborted",
"@session_user", "@assigns", "@session", "@params", "@request_origin", "@variables_added",
"@request", "@__bp_line", "@users"]

irb(#<UsersController:0xb7735e4c>):012:0> local_variables
=> ["_"]

irb(#<UsersController:0xb7735e4c>):013:0> request.request_uri
=> "/users/list"

irb(#<UsersController:0xb7735e4c>):014:0> controller_name
=> "users"

```

Tudo o que está disponível no ambiente pode ser acessado da linha de comando e código Ruby arbitrário pode ser executado também. Finalmente, para sair, use o comando *exit*, que retorna o controle à aplicação. Múltiplos *breakpoints* podem ser usados para paradas seqüenciais. Nesse caso, *exit* moverá a execução para o próximo *breakpoint*.

Como podemos ver, a depuração de aplicações é tão simples em Rails quanto qualquer outra tarefa.

ASSOCIAÇÕES POLIMÓRFICAS

Um tipo de associação que não tivemos oportunidade de explorar até o momento são associações polimórficas. Esse tipo de associação é mais uma das novidades que foram introduzidas no Rails 1.1 e são responsáveis por uma facilidade muito maior em certos tipos de aplicações.

Vamos supor que, em nossa aplicação, tivéssemos a necessidade de saber sobre cada modificação feita a um registro, ou seja, auditar cada operação de persistência ao banco de dados. Essa é uma necessidade que eu já encontrei em algumas aplicações, especialmente requisitada pelo usuário.

Para armazenar esses dados, poderíamos criar um tabela que conteria o data da operação, o tipo de operação, o usuário que a realizou e o objeto sobre o qual a mesma foi realizada. No caso aqui, para demonstrar as nossas associações, vamos ignorar um detalhe óbvio: a auditoria da remoção do objeto que, pelo método que vamos empregar, não funcionaria, já que o objeto está sendo removido do banco de dados e qualquer associação com o mesmo se tornaria inválida nesse instante.

Ignorando isso, a primeira coisa precisamos fazer é criar a tabela que conterá nossos registros de criação e

atualização. Vamos criar o modelo de dados diretamente já que não precisamos, no momento, exibir as auditorias:

```
ronaldo@minerva:~/tmp/gtd$ script/generate model audit
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/audit.rb
create test/unit/audit_test.rb
create test/fixtures/audits.yml
exists db/migrate
create db/migrate/011_create_audits.rb
```

Depois de criado o modelo, editamos a migração:

```
class CreateAudits < ActiveRecord::Migration
  def self.up
    create_table :audits do |t|
      t.column :operation, :string
      t.column :recorded_at, :datetime
      t.column :auditable_id, :integer
      t.column :auditable_type, :string
      t.column :user_id, :string
    end
  end

  def self.down
    drop_table :audits
  end
end
```

O significado das duas colunas `auditable_id` e `auditable_type` se tornará claro em um instante.

Com o modelo em mãos, podemos editá-lo:

```
class Audit < ActiveRecord::Base
  belongs_to :auditable, :polymorphic => true
end
```

Essa linha é importante. Ela indica a interface polimórfica que vamos adotar para nossa aplicação. O nome é arbitrário mas, uma vez escolhido, será usado na indicação da associações.

Escolhendo um modelo qualquer, podemos fazer o seguinte agora:

```
class Context < ActiveRecord::Base
  attr_protected :user_id
  validates_presence_of :name
  validates_uniqueness_of :name
  validates_presence_of :user_id
```

```

has_many :actions
has_many :audits, :as => :auditable

end

```

A definição da associação agora possui um novo parâmetro que indica a interface polimórfica que definimos.

E agora, podemos usar a associação, pelo console:

```

ronaldo@minerva:~/tmp/gtd$ script/console
Loading development environment.

>> context = Context.find(1)
=> #<Context:0xb7800764 @attributes={"name"=>"@Home", "id"=>"1", "user_id"=>"1"}>

>> context.audits
=> []

>> context.audits.create(:operation => "insert", :recorded_at => DateTime.now)
=> #<Audit:0xb77ea70c @new_record=false, @errors=#<ActiveRecord::Errors:0xb77e5c5c
@base=#<Audit:0xb77ea70c ...>, @errors={}, @attributes={"recorded_at"=>#<DateTime:
106012954818286597/43200000000,-1/8,2299161>, "auditable_type"=>"Context", "id"=>1,
"auditable_id"=>1, "operation"=>"insert"}>

>> context.audits[0].auditable_id
=> 1

>> context.audits[0].auditable_type
=> "Context"

```

O significado das colunas se torna claro agora: a coluna `auditable_id` contém o identificador do registro relacionado à associação—no caso acima, o contexto com o `id` cujo valor é 1—e a coluna `auditable_type` contém o nome da classe relacionada—no caso acima, `Context`.

Vamos testar com outro modelo:

```

class Project < ActiveRecord::Base

  validates_presence_of :name
  validates_uniqueness_of :name

  has_many :actions
  has_many :contexts, :through => :actions, :select => "distinct contexts.*"
  has_many :audits, :as => :auditable

end

```

Ainda no console, podemos fazer:

```

>> reload!
Reloading...
=> [Audit, ApplicationController, Context, Project, Action, Resource]

>> project = Project.find(1)

```

```

=> #<Project:0xb7786720 @attributes={"name"=>"Build a House", "id"=>"1", "description"=>"",
"user_id"=>"1", "active"=>"1"}>

>> project.audits
=> []

>> project.audits.create(:operation => "insert", :recorded_at => DateTime.now)
=> #<Audit:0xb7776960 @new_record=false, @errors=#<ActiveRecord::Errors:0xb776eb48
@base=#<Audit:0xb7776960 ...>, @errors={}, @attributes={"recorded_at"=>#<DateTime:
21202590990734567/8640000000,-1/8,2299161>, "auditable_type"=>"Project", "id"=>2, "auditable_id"=>1,
"operation"=>"insert"}>

>> project.audits[0].auditable_id
=> 1

>> project.audits[0].auditable_type
=> "Project"

```

Note o uso de `reload!` para recarregar os modelos recém-mudados. E veja que o registro criado para a nova classe que usamos muda o tipo da associação automaticamente. Qualquer outra classe que usássemos faria o mesmo. Selezionando os dados no MySQL, vemos os dados demonstrando isso claramente:

```

mysql> select * from audits;
+----+-----+-----+-----+-----+
| id | operation | recorded_at | auditable_id | auditable_type | user_id |
+----+-----+-----+-----+-----+
| 1 | insert | 2006-09-24 23:13:59 | 1 | Context | NULL |
| 2 | insert | 2006-09-24 23:14:04 | 1 | Project | NULL |
+----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

```

Como podemos ver, associações polimórficas tem esse nome porque elas são capazes de representar relações para objetos diferentes por meio de colunas especiais que recebem seus valores automaticamente. Essas colunas são usadas pelo Rails para gerar declarações SQL específicas, baseadas na classe que está fazendo a chamada, transparentemente para a aplicação.

Uma vez que temos essas associações, podemos fazer nossa auditoria de maneira automática. Para isso, vamos utilizar módulos para incorporar essa funcionalidade aos modelos.

MÓDULOS

Para usar módulos nesse caso, vamos criar um arquivo específico no diretório `extras`, chamado `auditing.rb`:

```

module GTD
  module Auditing
    def self.append_features(base)
      base.after_create do |object|
        object.audits.create(:operation => "insert",
          :recorded_at => DateTime.now)
      end
    end
  end
end

```

```

    end

    base.after_update do |object|
      object.audits.create(:operation => "update",
        :recorded_at => DateTime.now)
    end

  end

end

```

O que esse arquivo faz é definir um módulo que será incorporado aos modelos na aplicação. Para evitar colisões de nomes, criamos o módulo dentro de outro, específico para nossa aplicação.

O método `append_features`, inerente a módulos, é rodado automaticamente pelo Ruby quando um módulo é adicionado a uma classe. No caso acima, usamos o método para adicionar dois filtros às classes em questão: um após a criação e um após cada operação de salvamento. Esses métodos simplesmente adicionam mais um registro de auditoria aos existentes.

Precisamos agora mudar o arquivo `environment.rb` para requerer o arquivo:

```

# Be sure to restart your web server when you modify this file.

# Uncomment below to force Rails into production mode when
# you don't control web/app server and can't set it the proper way
# ENV['RAILS_ENV'] ||= 'production'

# Specifies gem version of Rails to use when vendor/rails is not present
RAILS_GEM_VERSION = '1.1.6'

# Bootstrap the Rails environment, frameworks, and default configuration
require File.join(File.dirname(__FILE__), 'boot')

Rails::Initializer.run do |config|
  # Settings in config/environments/* take precedence those specified here

  # Skip frameworks you're not going to use (only works if using vendor/rails)
  # config.frameworks -= [ :action_web_service, :action_mailer ]

  # Add additional load paths for your own custom dirs
  config.load_paths += %W( #{RAILS_ROOT}/extras )

  # Force all environments to use the same logger level
  # (by default production uses :info, the others :debug)
  # config.log_level = :debug

  # Use the database for sessions instead of the file system
  # (create the session table with 'rake db:sessions:create')
  # config.action_controller.session_store = :active_record_store

  # Use SQL instead of Active Record's schema dumper when creating the test database.
  # This is necessary if your schema can't be completely dumped by the schema dumper,
  # like if you have constraints or database-specific column types
  # config.active_record.schema_format = :sql

  # Activate observers that should always be running
  # config.active_record.observers = :cacher, :garbage_collector

  # Make Active Record use UTC-base instead of local time

```

```

# config.active_record.default_timezone = :utc

# See Rails::Configuration for more options
end

# Add new inflection rules using the following format
# (all these examples are active by default):
# Inflector.inflections do |inflect|
#   inflect.plural /^(ox)$/i, '\1en'
#   inflect.singular /^(ox)en/i, '\1'
#   inflect.irregular 'person', 'people'
#   inflect.uncountable %w( fish sheep )
# end

# Include your application configuration below
require "user_filter.rb"
require "auditing.rb"

ActionMailer::Base.server_settings =
{
  :address => "mail.yourdomain.com",
  :domain => "yourdomain.com",
  :user_name => "user@yourdomain.com",
  :password => "*****",
  :authentication => :login
}

```

E por fim, adicionamos o módulo em um de nossos modelos:

```

class Context < ActiveRecord::Base

  include GTD::Auditing

  attr_protected :user_id

  validates_presence_of :name
  validates_uniqueness_of :name

  validates_presence_of :user_id

  has_many :actions
  has_many :audits, :as => :auditable

end

```

Agora, para qualquer operação feita nesse modelo de dados, uma trilha de auditoria será gravada. Podemos comprovar isso, salvando um registro existente e criando um novo, o que resulta, em nosso banco, nos seguintes registros:

```

mysql> select * from audits;
+----+-----+-----+-----+-----+
| id | operation | recorded_at | auditable_id | auditable_type | user_id |
+----+-----+-----+-----+-----+
| 1 | insert | 2006-09-24 23:13:59 | 1 | Context | NULL |
| 2 | insert | 2006-09-24 23:14:04 | 1 | Project | NULL |
| 3 | update | 2006-09-24 23:19:03 | 1 | Context | NULL |
| 4 | insert | 2006-09-24 23:24:58 | 12 | Context | NULL |
+----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

Uma modificação que podemos fazer em nosso módulo é permitir automaticamente a criação de uma

associação:

```
module GTD

  module Auditing

    def self.append_features(base)

      base.after_create do |object|
        object.audits.create(:operation => "insert",
                             :recorded_at => DateTime.now)
      end

      base.after_update do |object|
        object.audits.create(:operation => "update",
                             :recorded_at => DateTime.now)
      end

      base.has_many :audits, :as => :audit
    end
  end
end
```

Isso permite que eliminemos a mesma chamada de cada classe em que usarmos o módulo. Modificar o resto das classes para usar o módulo é um exercício para o leitor.

Obviamente, no código acima, falta uma coisa: registrar o usuário responsável pelas chamadas. A princípio, poderíamos pensar em simplesmente usar o atributo `user_id` presente em várias classes. Mas isso pode não ser válido para uma situação: o que acontece com classes que não tem esse atributo? No caso de nossa aplicação, só temos uma, que é o modelo de dados do próprio usuário e, nesse caso não podemos utilizar o `id` dos registros da mesma porque não a ver com usuário que está fazendo a modificação. Não podemos usar também uma variável de sessão—que seria talvez o método mais óbvio—pois sessões não estão acessíveis a modelos.

A solução é usar um estratégia especial. Existe um local que seria acessível a todos os modelos: uma variável de classe que todas outras classes podem acessar. Para isto, vamos modificar o modelo de dados de usuários, que é o local mais propício para isso:

```
class User < ActiveRecord::Base

  cattr_accessor :current_user_id

  validates_presence_of :name
  validates_presence_of :login
  validates_presence_of :password
  validates_presence_of :email

  validates_uniqueness_of :login

end
```

Essa variável pode ser acessada diretamente na classe e usada por qualquer modelo. Agora, precisamos

atribuir um valor à mesma antes de qualquer uso das classes. Podemos fazer isso em nosso *controller* raiz:

```
class ApplicationController < ActionController::Base

  before_filter :authenticate
  before_filter :set_current_user_id

  skip_before_filter :authenticate_administration, :only => [:access_denied]

  around_filter UserFilter.new(Context, :needed_scopes)
  around_filter UserFilter.new(Project, :needed_scopes)
  around_filter UserFilter.new(Action, :needed_scopes)
  around_filter UserFilter.new(Resource, :needed_scopes)

  helper_method :session_user

  def access_denied
    render :template => "shared/access_denied"
  end

  protected

  def set_current_user_id
    User.current_user_id = session[:user]
  end

  def session_user
    @session_user ||= User.find(:first, :conditions => ['id = ?', session[:user]])
  end

  def authenticate
    unless session[:user]
      session[:return_to] = request.request_uri
      redirect_to :controller => "login", :action => "login"
      return false
    end
    return true
  end

  def authenticate_administration
    unless session_user && session_user.admin?
      redirect_to :action => "access_denied"
      return false
    end
    return true
  end

  def needed_scopes(target_class)
  {
    :find => { :conditions => ["#{target_class.table_name}.user_id = ?", session[:user]] },
    :create => { :user_id => session[:user] }
  }
  end

end
```

Com isso feito, podemos fazer a modificação em nosso módulo de auditoria:

```
module GTD

  module Auditing

    def self.append_features(base)

      base.after_create do |object|
        object.audits.create(:operation => "insert",
```

```

    :recorded_at => DateTime.now,
    :user_id => User.current_user_id)
end

base.after_update do |object|
  object.audits.create(:operation => "update",
    :recorded_at => DateTime.now,
    :user_id => User.current_user_id)
end

base.has_many :audits, :as => :auditible

end

end

```

Recarregando o nosso servidor e executando alguma operação em nossas classes auditadas, teremos o seguinte em nosso banco de dados:

```

mysql> select * from audits;
+----+-----+-----+-----+-----+-----+
| id | operation | recorded_at | auditible_id | auditible_type | user_id |
+----+-----+-----+-----+-----+-----+
|  1 | insert   | 2006-09-24 23:13:59 |      1 | Context     | NULL       |
|  2 | insert   | 2006-09-24 23:14:04 |      1 | Project     | NULL       |
|  3 | update   | 2006-09-24 23:19:03 |      1 | Context     | NULL       |
|  4 | insert   | 2006-09-24 23:24:58 |     12 | Context     | NULL       |
|  5 | update   | 2006-09-24 23:26:35 |     12 | Context     | NULL       |
|  6 | update   | 2006-09-24 23:40:42 |     12 | Context     |      1     |
|  7 | insert   | 2006-09-24 23:40:56 |     13 | Context     |      1     |
+----+-----+-----+-----+-----+-----+
7 rows in set (0.00 sec)

```

Essa é outra técnica que podemos usar em nossos módulos para acessar objetos ou valores que precisamos. Deve ser usada com cuidado, pois pode gerar código espaguete, mas é uma maneira fácil e interessante de transferir dados de acordo com a necessidade.

Ajax

Uma das grandes tendências atuais no desenvolvimento de aplicações Web é o que se tornou conhecido como Web 2.0, que é, nada mais do que uma tentativa de aproximar as aplicações Web de suas contrapartes *desktop*. Aplicações feitas no estilo da assim chamada Web 2.0 possuem características de integração e usabilidade muito superiores às aplicações que vinham sendo comumente desenvolvidas e estão se transformando em uma referência para o que deve ou não deve ser feito na Internet.

Uma das grandes marcas desse modelo de desenvolvimento são aplicações que usam novas maneiras de recuperar dados no servidor, baseadas em JavaScript, para reduzir a necessidade de recarregamento de páginas tão tradicional em aplicações Web. E, embora essa não seja a principal característica de um aplicação Web 2.0, isso é o que se tornou um dos fatores dominantes na definição do que seria essa nova Internet.

Dentro dessa definição, a popularização de ferramentas baseadas na técnica conhecida como Ajax—da qual você com certeza já ouviu falar e, muito provavelmente já utilizou—foi um dos fatores predominantes na aceitação dessas novas aplicações. Marca, mais do que funcionalidade, predominou inicialmente.

Para aqueles que nunca ouviram falar em Ajax, um das definições mais comuns é a do próprio nome, que originalmente seria uma abreviação de *Asynchronous JavaScript and XML*. Embora seja pouco mais do que um uso avançado do que era conhecido como DHTML, o aumento óbvio do uso da técnica se deve ao fato de que os navegadores mais usados no mercado estabilizaram suas interfaces JavaScript, HTML, XML e CSS ao ponto de que desenvolver uma aplicação usando técnicas avançadas não é mais uma questão de desenvolver versões diferentes da mesma coisa.

Um dos grandes motivos da popularização do Rails nos últimos tempos tem sido o suporte do mesmo a algumas dessas bibliotecas que, embora não seja as mais avançadas e completas, estão tão integradas no *framework* que usar as mesmas se tornou algo assumido e fácil o suficiente para beneficiar mesmo o desenvolvedor iniciante.

Em nosso tutorial vamos agora explorar algumas dessas integrações mostrando como o Ajax pode ser realmente utilizado para melhorar as nossas aplicações, dando-lhes não só uma usabilidade melhor como uma aparência de melhor tempo de resposta.

Obviamente, como qualquer outra técnica, o Ajax tem suas vantagens e desvantagens. Uma das principais dificuldades, por exemplo, é manter a acessibilidade da aplicação, ou seja, não sacrificar aqueles usuários com limitações de acesso, seja quais forem essas limitações. O objetivo do tutorial, nesse ponto não é mostrar como resolver essas dificuldades. Procuramos aqui apenas mostrar rapidamente como a técnica pode ser empregada. Recomendamos que o leitor que opte por empregar qualquer técnica relacionada em sua aplicação procure material que está plenamente disponível na Internet trabalhando esses problemas e apontando possíveis soluções.

Um dos outros grandes problemas do Ajax está no *upload* de arquivos, já que as requisições são feitas geralmente via JavaScript e limitações de tamanho, velocidade e principalmente de segurança, impedem que *uploads* sejam processados diretamente por trás das cenas. Existem soluções parciais para esses problemas e até pouco tempo atrás o Rails suportava uma delas como parte do código oficial. Em versões mais recentes do Rails, o suporte foi retirado do código principal e movido para um *plugin*, uma demonstração clara dos problemas ainda existentes com a técnica.

Problemas à parte, a técnica é extremamente útil e suas aplicações poderão ganhar muito usando mesmo as implementações mais básicas da mesma. Como mencionado acima, essa é a nossa proposta aqui: mostrar o que pode ser feito e permitir que o leitor evolua suas próprias soluções, aprendendo a lidar com os problemas no uso prático da técnica.

Depois de todos esses comentários e ressalvas, podemos partir para nossa implementação. Para isto, vamos

aproveitar a nossa página *home* que até o momento estava convenientemente vazia. Na metodologia GTD, a coisa mais importante é a próxima ação, ou seja, quais ações você ainda precisa completar. Faz sentido que a nossa página *home* seja um modo de visualizar e trabalhar rapidamente com essas ações.

Para começar, então, vamos simplesmente exibir nossas ações na página principal, separadas pelo contexto. Voltando ao nosso *controller*, teríamos o seguinte:

```
class HomeController < ApplicationController

  def index
    @actions = Action.find(:all,
      :conditions => "done = 0",
      :include => [:user, :project, :context],
      :order => "actions.description")
  end

end
```

Recuperamos as ações ainda não completadas. Note que o código acima assume que você alterou o modelo de ações conforme mostrado abaixo, criando uma associação entre usuário e ação:

```
class Action < ActiveRecord::Base

  attr_protected :created_at, :completed_at

  def done=(value)
    if ActiveRecord::ConnectionAdapters::Column.value_to_boolean(value)
      self.completed_at = DateTime.now
    else
      self.completed_at = nil
    end
    write_attribute("done", value)
  end

  belongs_to :context
  belongs_to :project
  belongs_to :user

  validates_presence_of :description

  validates_presence_of :context_id
  validates_presence_of :project_id

  has_and_belongs_to_many :resources, :order => "name"
end
```

Agora que temos nossas ações, podemos criar a nossa *view* separando as ações por contexto:

```
<h1>Next Actions</h1>

<% @actions.group_by(&:context).each do |context, actions| %>
<div class="group">

  <h2><%= context.name %></h2>

  <ul>
```

```

<% actions.each do |action| %>
  <li><%= action.description %></li>
<% end %>
</ul>

</div>
<% end %>

```

Na *view* acima estamos usando uma técnica interessante para separados as ações por contexto: o método `group_by`. Esse método agrupa um enumerado qualquer (como nossas ações acima) em conjuntos cujo critério de separação é a mensagem passada como parâmetro.

Estudar o método `group_by`, no código do Rails, é um bom exercício para entender o poder do Ruby. A implementação depende de usos tão sofisticados e elegantes da linguagem que só o estudo e entendimento da técnica usada será suficiente para avançar razoavelmente seu conhecimento da linguagem. O método foi inserido diretamente no módulo que controla enumerações no Ruby, passando a funcionar para qualquer tipo de enumeração que venha a ser definida em código. O parâmetro passado é também uma adição do Rails, modificando a classe `Symbol` do Ruby. Em uma linguagem estática isso simplesmente não seria possível ou seria muito difícil de implementar de maneira tão simples.

Voltando ao código acima, estamos separando as nossas ações pelo seu contexto. O método `group_by` retorna uma tabela *hash* separada por contexto que pode ser usada em um *loop*, como fazemos acima. Essa tabela *hash* possui como chaves o resultado da mensagem que passamos como parâmetro (no caso acima, o contexto) e como valor um *array* dos objetos que filtramos (no caso acima, as ações).

O resultado inicial é:

Como podemos ver, o método é extremamente útil, permitindo uma separação dos dados de maneira bem clara e simplificada. Essa é uma das vantagens de usar uma linguagem dinâmica como o Ruby.

Agora, adicionando o fragmento de CSS abaixo ao nosso arquivo `default.css`, melhoramos a nossa apresentação:

```
div.group
{
  border: 1px solid #ccc;
  margin-bottom: 10px;
}

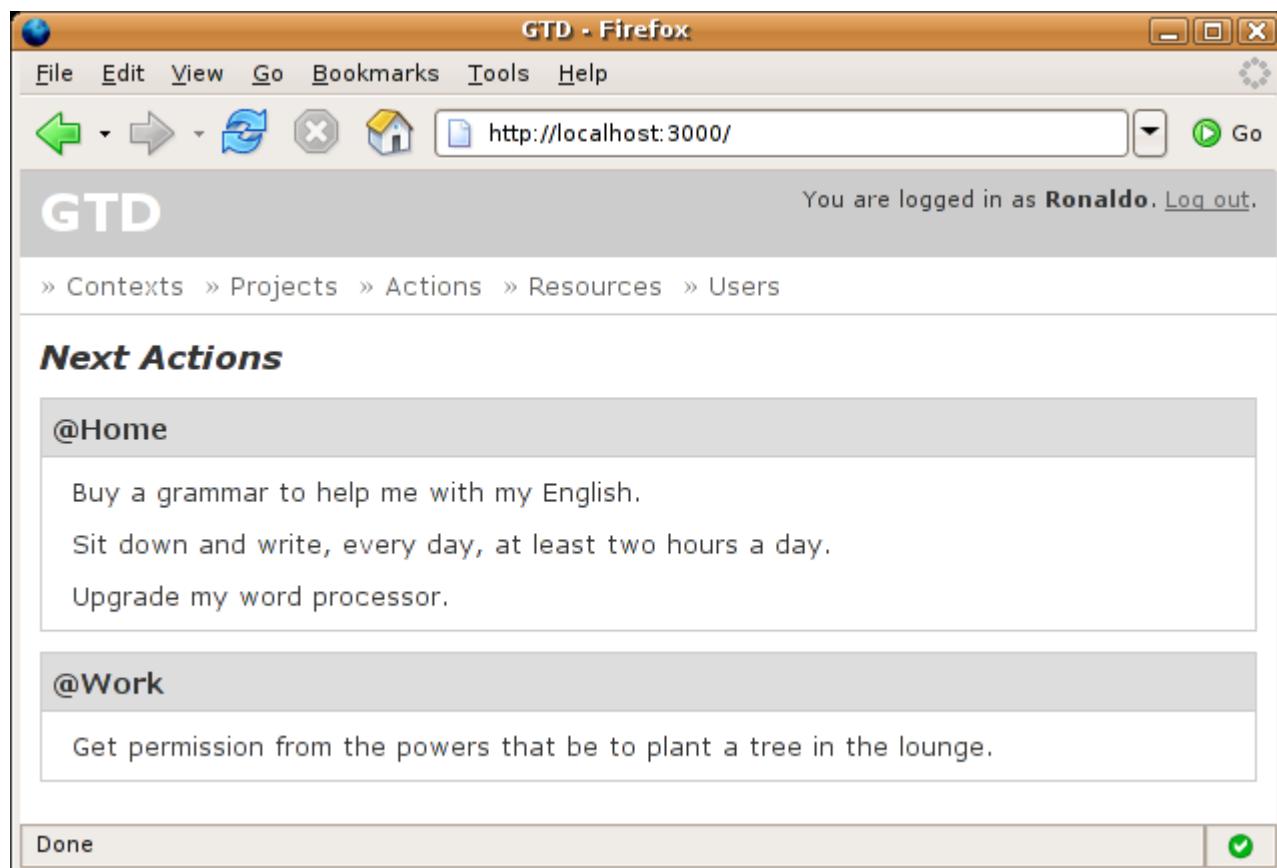
div.group h2
{
  padding: 5px;
  margin: 0px;
  border-bottom: 1px solid #ccc;
  background-color: #ddd;
  font-size: 110%;
}

div.group ul
{
  list-style: none;
  margin: 0;
  padding: 0;
}

div.group ul li
{
```

```
    margin: 8px 15px;  
}
```

O resultado é o seguinte:



Temos agora uma apresentação básica para trabalharmos. A primeira coisa que queremos é uma maneira de marcar ações como completadas. Seria interessante para isso exibir pelo menos algumas das ações já completadas para termos uma referência. Podemos fazer isso, modificando o nosso *controller*:

```
class HomeController < ApplicationController  
  
  def index  
    @actions = Action.find(:all,  
      :conditions => "done = 0",  
      :include => [:user, :project, :context],  
      :order => "actions.description")  
    @completed_actions = Action.find(:all,  
      :conditions => "done = 1",  
      :include => [:user, :project, :context],  
      :order => "actions.description"),  
      :limit => 3  
  end  
end
```

Podemos também melhorar o nosso código acima refatorando as chamadas em um método mais flexível, diretamente no modelo de dados. Um exemplo disso seria:

```
class Action < ActiveRecord::Base

attr_protected :created_at, :completed_at

def done=(value)
  if ActiveRecord::ConnectionAdapters::Column.value_to_boolean(value)
    self.completed_at = DateTime.now
  else
    self.completed_at = nil
  end
  write_attribute("done", value)
end

belongs_to :context
belongs_to :project
belongs_to :user

validates_presence_of :description

validates_presence_of :context_id
validates_presence_of :project_id

has_and_belongs_to_many :resources, :order => "name"

def self.find_by_status(done, limit = nil)
  find :all,
    :conditions => ["done = ?", done ? 1 : 0],
    :include => [:user, :project, :context],
    :order => "actions.description",
    :limit => limit
end

end
```

Voltando ao nosso *controller*, isso daria:

```
class HomeController < ApplicationController

def index
  @actions = Action.find_by_status(false)
  @completed_actions = Action.find_by_status(true, 3)
end

end
```

Depois, modificando nossa view, temos:

```
<h1>Next Actions</h1>

<% @actions.group_by(&:context).each do |context, actions| %>
<div class="group">

<h2><%= context.name %></h2>

<ul>
<% actions.each do |action| %>
  <li><%= action.description %></li>
```

```

<% end %>
</ul>

</div>
<% end %>

<div class="group">
  <h2>Completed Actions</h2>
  <ul>
    <% @completed_actions.each do |action| %>
      <li><%= action.description %></li>
    <% end %>
  </ul>
</div>

```

O resultado final é o seguinte:

The screenshot shows a Firefox browser window with the title "GTD - Firefox". The address bar displays "http://localhost:3000/". The page content is as follows:

- GTD** (logged in as Ronaldo, [Log out](#))
- Navigation: » Contexts » Projects » Actions » Resources » Users
- Next Actions**

 - @Home**
 - Buy a grammar to help me with my English.
 - Sit down and write, every day, at least two hours a day.
 - Upgrade my word processor.
 - @Work**
 - Plant the tree in the lounge.
 - Completed Actions**
 - Get permission from the powers that be to plant a tree in the lounge.

- Done** (with a checked checkbox)

Agora, podemos introduzir alguma forma de marcar se a ação está completada ou não:

```

<h1>Next Actions</h1>

<% @actions.group_by(&:context).each do |context, actions| %>
<div class="group" id="<%= context.id %>">

  <h2><%= context.name %></h2>

  <ul>
    <% actions.each do |action| %>
      <li id="context_action_<%= action.id %>">
        <%= check_box_tag "completed[#{action.id}]", action.id, false, :onclick =>
        "#{remote_function(:url => { :action => "mark", :id => action })}"  %>
        <%= action.description %>
      </li>
    <% end %>
  </ul>

</div>
<% end %>

<div class="group">

  <h2>Completed Actions</h2>

  <ul id="completed_actions">
    <% @completed_actions.each do |action| %>
      <li><%= action.description %></li>
    <% end %>
  </ul>

</div>

```

O significado de todos esses *ids* conferidos a elementos ficará aparente em breve. Por hora, o importante é a chamada remota que vamos fazer. Repare o fragmento de código que introduzimos:

```

<%= check_box_tag "completed[#{action.id}]", action.id, false,
:onclick => "#{remote_function(:url => "mark", :id => action) }"  %>

```

Esse fragmento declara um elemento de formulário do tipo *checkbox*, desmarcado e com um evento *onclick* definido para uma ação em JavaScript. Aqui entra o Ajax integrado ao Rails através de uma biblioteca chamada Prototype. Antes de podermos fazer executar o código acima, precisamos incorporar essa biblioteca à nossa aplicação. Por padrão, no momento em que o esqueleto da aplicação foi gerado, essas biblioteca (entre outras) foi colocada no diretório `public/javascripts`, como você pode ver visualizando o diretório.

O que precisamos é somente invocá-las, o que conseguimos com uma pequena mudança em nosso *layout* base:

```

<html>

<head>
  <title>GTD</title>
  <%= stylesheet_link_tag "default" %>
  <%= stylesheet_link_tag "scaffold" %>
  <%= javascript_include_tag :defaults %>
</head>

```

```

<body>

  <h1 id="header">GTD</h1>

  <% if session_user %>
  <p id="login-information">
    You are logged in as <strong><%= session_user.name %></strong>.
    <%= link_to "Log out", :controller => "login", :action => "logout" %>.
  </p>
  <% end %>

  <ul id="menu">
    <li><%= link_to_unless_current "&raquo; Contexts", :controller => "contexts", :action => "list"
  %></li>
    <li><%= link_to_unless_current "&raquo; Projects", :controller => "projects", :action => "list"
  %></li>
    <li><%= link_to_unless_current "&raquo; Actions", :controller => "actions", :action => "list"
  %></li>
    <li><%= link_to_unless_current "&raquo; Resources", :controller => "resources", :action => "list"
  %></li>
    <% if session_user && session_user.admin? %>
      <li><%= link_to_unless_current "&raquo; Users", :controller => "users", :action => "list" %></li>
    <% end %>
  </ul>

  <div id="contents"><%= yield %></div>

</body>

</html>

```

O resultado dessa chamada, que pode ser visto visualizando-se a código fonte da página gerada, é adicionar uma série de bibliotecas JavaScript à página. Somente faça a invocação acima se realmente precisar já que elas representam quase 200KB de JavaScript que teriam que ser baixados e interpretados pelo navegador.

Obviamente, se o seu servidor estiver configurado apropriadamente, os arquivos somente ser baixados uma única vez, sem causar mais do que uma demora na carga inicial da página. Mesmo assim, é interessante ter um certo cuidado em usar as bibliotecas—e quaisquer outras que forem necessárias—para não gerar páginas que demoram demais a ser carregadas e cuja usabilidade é potencialmente reduzida.

A velocidade de interpretação de JavaScript varia de navegador para navegador e pode ser realmente lenta em alguns, mesmo entre os mais usados. Por isso, quanto menos JavaScript carregado de uma vez, melhor. Assim, se for possível, ao invés de chamar o código acima, introduza cada biblioteca de acordo com a necessidade e não as carregue em páginas onde não forem necessárias.

A nossa página agora está assim:

Se você clicar em um dos *checkboxes* acima verá que nada acontece—externamente. Se você consultar o *log* da aplicação verá que uma ação está sendo chamada por trás das cenas—e obviamente falhando porque não a definimos ainda. Você pode também visualizar o código da página para ver o código que a chamada ao método `remote_function` gerou.

O que essa função faz, basicamente, é invocar uma ação que passamos como parâmetro. Ela não retorna valor a menos que implementemos a ação:

```
class HomeController < ApplicationController

  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
  end

  def mark
    @action = Action.find(params[:id])
    @action.update_attribute("done", true)
  end

```

```
end
```

A ação continua não retornando coisa alguma, mas agora, se você clicar em uma das ações e recarregar a página, verá que ela despareceu do seu contexto e agora está na lista de ações completadas.

O objetivo agora é fazer a transição da ação visualmente de uma lista para a outra. Vamos começar começar fazendo a ação desaparecer. Para isto, vamos usar pela primeira vez um novoceade no Rails 1.1 que são os *templates RJS* (Rails JavaScript), *views* descrevendo comandos JavaScript. Essas *views*, não surpreendentemente, funcionam de maneira similar às que descrevem elementos XML.

Para usar uma *view* contendo um *template RJS*, vamos criar o arquivo da *view* da ação *mark*, chamado *mark.rjs* no diretório *app/views/home*:

```
page.remove "context_action_#{@action.id}"
```

Como você pode ver, o *template* é bem parecido com o que usamos anteriormente para gerar XML. A diferença é que esse gera comandos JavaScript que são devolvidos para a aplicação e interpretados assim que são retornados. No caso acima, um comando é gerado para remover o elemento especificado pela *id* passado ao método *remove* do objeto *page*, que é automaticamente definido para esse tipo de *templates*.

Se você executar a ação, verá que o elemento agora desaparece do seu contexto. Precisamos fazê-lo agora aparecer em nossa lista de ações completadas. Existem duas maneiras de fazer isso: primeiro, podemos simplesmente criar um novo elemento na lista com a descrição da ação; e, segundo, podemos renderizar somente a lista de ações completadas novamente. O segundo método pode parecer mais custoso—e realmente é—mas tem a vantagem de permitir, entre outras coisas, mantermos a nossa apresentação consistente, como veremos logo abaixo.

Vamos dizer que, eventualmente, queiramos exibir mais do que somente a descrição da ação em nossas listas. Se geramos simplesmente as nossas listas em cada lugar que precisamos, veremos que toda vez que fizermos uma mudança em nossa apresentação, teremos que modificar todos locais que exibem a lista. O que podemos fazer é usar *partials* para resolver esse problema. Vamos mudar a nossa *view* principal para:

```
<h1>Next Actions</h1>

<% @actions.group_by(&:context).each do |context, actions| %>
<div class="group" id="context_<%= context.id %>">

  <h2><%= context.name %></h2>

  <ul>
    <% actions.each do |action| %>
      <li id="context_action_<%= action.id %>">
        <%= check_box_tag "completed[#{action.id}]", action.id, false, :onclick =>
        "#{remote_function(:url => { :action => "mark", :id => action })}" %>
        <%= action.description %>
    <% end %>
  </ul>
<% end %>
```

```

        </li>
    <% end %>
</ul>

</div>
<% end %>

<div class="group">
    <h2>Completed Actions</h2>
    <%= render :partial => "completed_actions", :object => @completed_actions %>
</div>

```

Precisamos agora do *partial*, cujo arquivo será `_completed_actions.rhtml`:

```

<ul id="completed_actions">
<% @completed_actions.each do |action| %>
    <li><%= action.description %></li>
<% end %>
</ul>

```

O resultado visual é o mesmo, mas há uma grande vantagem. Podemos agora atualizar a nossa lista dinamicamente. Primeiro, mudamos a ação *mark* em nosso *controller*:

```

class HomeController < ApplicationController

  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
  end

  def mark
    @action = Action.find(params[:id])
    @action.update_attribute("done", true)
    @completed_actions = Action.find_by_status(true, 3)
  end

end

```

E, depois disso, nosso *template RJS*:

```

page.remove "context_action_#{@action.id}"
page.replace "completed_actions", :partial => "completed_actions", :object => @completed_actions

```

A segunda linha acima renderiza o *partial* que criamos (já podemos ver aqui os benefícios dessa separação ao podermos reusar o código) e substitui o elemento HTML informado no primeiro parâmetro pelo resultado da renderização.

O resultado, após clicar em nossa primeira ação mostrada na tela anterior é:

The screenshot shows a Firefox browser window titled "GTD - Firefox". The address bar displays "http://localhost:3000/". The page content is as follows:

You are logged in as **Ronaldo**. [Log out.](#)

» Contexts » Projects » Actions » Resources » Users

Next Actions

@Home

- Sit down and write, every day, at least two hours a day.
- Upgrade my word processor.

@Work

- Plant the tree in the lounge.

Completed Actions

- Buy a grammar to help me with my English.
- Get permission from the powers that be to plant a tree in the lounge.

Done

Como podemos ver, o resultado da ação, embora acesse o servidor e funcione dentro do modelo normal do Rails, em nenhum momento recarrega a página. Temos uma situação bem mais prática para o usuário. Pela aplicação da mesma técnica poderíamos criar uma ação reversa *unmark* e remover um elemento da lista de completadas, retornando-o ao seu contexto.

Um probleminha que temos em nosso código é o que acontece quando remover a última ação de um contexto, deixando somente o título do contexto visível na página.

O resultado é visualmente desagradável como podemos ver abaixo:

Podemos resolver isso com pequenas modificações. Primeiro, em nosso *controller*:

```
class HomeController < ApplicationController

  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
  end

  def mark
    @action = Action.find(params[:id])
    @action.update_attribute("done", true)
    @completed_actions = Action.find_by_status(true, 3)
    @remove_context = @action.context.actions.count("done = 0") == 0
  end

end
```

Depois, em nosso *template* RJS:

```
page.remove "context_action_#{@action.id}"
```

```

page.replace "completed_actions", :partial => "completed_actions", :object => @completed_actions

if @remove_context
  page.remove "context_#{@action.context.id}"
end

```

Estamos utilizando métodos novos do *ActiveRecord* em nossos exemplos. Caso você não tenha familiaridade com os mesmos (exemplos acima são `update_attribute` e `count`), consulte a documentação para informações mais detalhadas.

A mudança resolve nosso problema como você poderá ver testando mais uma vez a remoção da última ação de um contexto.

Uma outra coisa que é muito utilizada em aplicações Ajax são indicações visuais de que uma ação está acontecendo. A primeira forma de fazer isso é através de uma indicação de que a ação remota começou e terminou; a segunda é indicar o resultado da ação remota. Vamos experimentar com ambas.

Para a primeira precisamos de uma imagem de indicação. Escolha uma imagem animada qualquer e modifique o seu *layout* básico:

```

<html>

<head>
  <title>GTD</title>
  <%= stylesheet_link_tag "default" %>
  <%= stylesheet_link_tag "scaffold" %>
  <%= javascript_include_tag :defaults %>
</head>

<body>

  <h1 id="header">GTD</h1>

  <% if session_user %>
  <p id="login-information">
    You are logged in as <strong><%= session_user.name %></strong>.
    <%= link_to "Log out", :controller => "login", :action => "logout" %>.
  </p>
  <% end %>

  <%= image_tag("indicator.gif", :id => "indicator", :style => "float: right; margin: 5px; display: none") %>

  <ul id="menu">
    <li><%= link_to_unless_current "&raquo; Contexts", :controller => "contexts", :action => "list"
    %></li>
    <li><%= link_to_unless_current "&raquo; Projects", :controller => "projects", :action => "list"
    %></li>
    <li><%= link_to_unless_current "&raquo; Actions", :controller => "actions", :action => "list"
    %></li>
    <li><%= link_to_unless_current "&raquo; Resources", :controller => "resources", :action => "list"
    %></li>
    <% if session_user && session_user.admin? %>
      <li><%= link_to_unless_current "&raquo; Users", :controller => "users", :action => "list" %></li>
    <% end %>
  </ul>

  <div id="contents"><%= yield %></div>

```

```
</body>  
</html>
```

Estamos usando uma imagem chamada `indicator.gif`, que está no diretório `public/images`, que é a localização padrão gerada pelo método `image_tag`. Usando um pouco de estilo CSS, ajustamos a posição da imagem e a escondemos temporariamente.

Agora, modificamos a nossa `view`, no ponto em que a ação remota é chamada:

```
<h1>Next Actions</h1>  
  
<% @actions.group_by(&:context).each do |context, actions| %>  
  <div class="group" id="context_<%= context.id %>">  
  
    <h2><%= context.name %></h2>  
  
    <ul>  
      <% actions.each do |action| %>  
        <li id="context_action_<%= action.id %>">  
          <%= check_box_tag "completed[#{action.id}]", action.id, false, :onclick =>  
            "#{remote_function(:url => { :action => "mark", :id => action }, :before =>  
              %(Element.show("indicator")), :complete => %(Element.hide("indicator")))}" %>  
          <%= action.description %>  
        </li>  
      <% end %>  
    </ul>  
  
  </div>  
<% end %>  
  
<div class="group">  
  <h2>Completed Actions</h2>  
  
  <%= render :partial => "completed_actions", :object => @completed_actions %>  
</div>
```

O método que gera a chamada remota aceita parâmetros adicionais que nos permitem manipular a chamada. No caso aqui, estamos utilizando dois deles: `before`, que recebe uma função JavaScript que é executada exatamente antes da chamada remota; e `complete`, que recebe outra função JavaScript que é executada depois que a requisição termina, tenha ela sucedido ou não. Estamos usando uma forma especial de `strings` no Rails, usando `%()`, para evitar ter que ficar escapando aspas, sejam duplas ou simples. Na funções, usamos chamadas de classe JavaScript definida na biblioteca Prototype para, respectivamente, exibir e esconder o indicador de ação.

Se você executar a chamada agora, verá provavelmente que o indicador nem pisca. Isso acontece porque a chamada local é rápida demais. Para testar a chamada melhor, você pode colocar o seguinte código na ação:

```
class HomeController < ApplicationController
```

```

def index
  @actions = Action.find_by_status(false)
  @completed_actions = Action.find_by_status(true, 3)
end

def mark
  @action = Action.find(params[:id])
  @action.update_attribute("done", true)
  @completed_actions = Action.find_by_status(true, 3)
  @remove_context = @action.context.actions.count("done = 0") == 0
  sleep(2)
end

end

```

O resultado é um bom indicador de ação que, quando a rede estiver mais lenta, servirá perfeitamente para avisar o usuário de que sua requisição está sendo processado, que alguma coisa está acontecendo, como mostrado abaixo:

The screenshot shows a Firefox browser window titled "GTD - Firefox". The address bar displays "http://localhost:3000/". The main content area of the browser shows the GTD application's interface. At the top, it says "You are logged in as **Ronaldo**. [Log out](#)". Below this, the navigation path is shown as "» Contexts » Projects » Actions » Resources » Users". The main content area has a heading "Next Actions". Under this heading, there are three items listed with checkboxes:

- Buy a grammar to help me with my English.
- Sit down and write, every day, at least two hours a day.
- Upgrade my word processor.

 Below the "Next Actions" section is another section titled "@Work" which contains one item:

- Plant the tree in the lounge.

 Further down is a section titled "Completed Actions" which contains the text "Get permission from the powers that be to plant a tree in the lounge." At the very bottom of the browser window, there is a "Done" button with a checked checkbox.

Um outro tipo de indicador visual é exibir alguma marcação na própria página indicando o elemento que foi modificado. No caso de uma ação completada, por exemplo, podemos iluminá-la na página usando um efeito

de página definido na biblioteca script.aculo.us, que também vem com o Rails.

Precisamos mudar o nosso *partial* para associar um *id* a nossa ação. Sem isso, não seremos capazes de usar o objeto HTML no JavaScript de nosso *template*. Mudando o nosso *partial*, temos:

```
<ul id="completed_actions">
<% @completed_actions.each do |action| %>
  <li id="context_action_<%= action.id %>"><%= action.description %></li>
<% end %>
</ul>
```

E mudamos o nosso *template RJS*:

```
page.remove "context_action_#{@action.id}"
page.replace "completed_actions", :partial => @completed_actions
page.visual_effect :highlight, "context_action_#{@action.id}", :duration => 3.5

if @remove_context
  page.remove "context_#{@action.context.id}"
end
```

O resultado é:

The screenshot shows a Firefox browser window titled "GTD - Firefox". The address bar displays "http://localhost:3000/". The page content is the GTD application. At the top right, it says "You are logged in as Ronaldo. [Log out](#)". Below that is a navigation menu with links to "Contexts", "Projects", "Actions", "Resources", and "Users". The main content area has a heading "Next Actions". Under this heading is a section titled "@Home" containing three unchecked checkboxes: "Buy a grammar to help me with my English.", "Sit down and write, every day, at least two hours a day.", and "Upgrade my word processor.". Below this is a section titled "Completed Actions" containing the text "Get permission from the powers that be to plant a tree in the lounge." and "Plant the tree in the lounge." (which is highlighted with a yellow background). At the bottom left is a "Done" button with a checkmark icon.

A indicação acima permite que o usuário tenha mais uma confirmação de que algo aconteceu, e, mais do que isso, do que aconteceu.

Uma mudança rápida que podemos fazer agora, para facilitar o código futuro é compactar o nosso *partial* para se aplicar a qualquer situação em nossa página. A nossa *view* ficaria:

```
<h1>Next Actions</h1>

<% @actions.group_by(&:context).each do |context, actions| %>
<div class="group" id="context_<%= context.id %>">

  <h2><%= context.name %></h2>

  <%= render :partial => "actions", :object => actions %>

</div>
<% end %>

<div class="group">

  <h2>Completed Actions</h2>

  <%= render :partial => "actions", :object => @completed_actions %>

</div>
```

Mudamos também o nosso *partial*, renomeado para `_actions.rhtml`, ficaria assim:

```
<ul <% if actions.first.done? %>id="completed_actions"<% end %>>
  <% actions.each do |action| %>
    <li id="context_action_<%= action.id %>">
      <% unless action.done? %>
        <%= check_box_tag "completed[#{action.id}]", action.id, false, :onclick =>
          "#{remote_function(:url => { :action => "mark", :id => action }, :before =>
            %(Element.show("indicator")), :complete => %(Element.hide("indicator")))}" %>
      <% end %>
      <%= action.description %>
    </li>
  <% end %>
</ul>
```

E, por fim, nosso *template RJS*:

```
page.remove "context_action_#{@action.id}"
page.replace "completed_actions", :partial => "actions", :object => @completed_actions
page.visual_effect :highlight, "context_action_#{@action.id}", :duration => 3.5

if @remove_context
  page.remove "context_#{@action.context.id}"
end
```

Uma das vantagens de usarmos *partials*, mesmo que tenhamos que lidar com diferenças como é o nosso caso aqui, quando estamos lidando com ações ainda não executadas que são exibidas agrupadas e ações já executadas, que são exibidas sem grupo, é a possibilidade de manter o nosso código coeso, sem nos repetir.

Esse é um dos princípios do Ruby e do Rails, conhecido como DRY ou “**Don't Repeat Yourself**”—ou seja, faça qualquer coisa uma única vez; não copie nada. Esse é um princípio sólido de desenvolvimento que pode evitar muita dor de cabeça na manutenção da aplicação, principalmente por evitar a proliferação de *bugs* resultando dos infames usos de copiar e colar código.

Esse é também o princípio por trás do que fizemos acima, mudando um método de buscar para o modelo, para evitar repetições de código, mesmo que seja de mera condições de busca. É também é o princípio por trás dos filtros *around* que criamos e de muitas outras instâncias de código nesse tutorial. Finalmente, os *plugins* são uma expressão forte de DRY. DRY é algo que você deve manter como uma atitude permanente de análise do código.

Assim, se precisarmos mudar o nosso *partial*, como no exemplo abaixo, é que veremos a vantagem disso:

```
<ul <% if actions.first.done? %>id="completed_actions"<% end %>>
  <% actions.each do |action| %>
    <li id="context_action_<%= action.id %>">
      <% unless action.done? %>
        <%= check_box_tag "completed[#{action.id}]", action.id, false, :onclick =>
        "#{remote_function(:url => { :action => "mark", :id => action }, :before =>
        %(Element.show("indicator")), :complete => %(Element.hide("indicator")))}" %>
      <% end %>
      <%= action.description %><br>
      <span>
        <strong>Project:</strong> <%= action.project.name %>
        <% if action.done? %>
          <br><strong>Context:</strong> <%= action.context.name %>
          <br><strong>Completed At:</strong> <%= action.completed_at.strftime("%m/%d/%y") %>
        <% end %>
      </span>
    </li>
  <% end %>
</ul>
```

Com a mudança acima, e o seguinte fragmento de CSS em nosso arquivo `default.css`:

```
div.group ul li span
{
  font-size: 85%;
  display: block;
  margin-left: 2.2em;
}
```

Com isso temos:

The screenshot shows a Firefox browser window titled "GTD - Firefox". The address bar displays "http://localhost:3000/". The page content is the GTD application, showing the following structure:

- GTD** (header)
- You are logged in as **Ronaldo**. [Log out.](#)
- Navigation: » Contexts » Projects » Actions » Resources » Users
- Next Actions**
- @Home**
 - Buy a grammar to help me with my English.
Project: Write a Book
 - Sit down and write, every day, at least two hours a day.
Project: Write a Book
 - Upgrade my word processor.
Project: Write a Book
- @Work**
 - Plant the tree in the lounge.
Project: Plant a Tree
- Completed Actions**
 - Get permission from the powers that be to plant a tree in the lounge.
Project: Plant a Tree
Context: @Work
Completed At: 09/25/06
- Buttons: Done (left) and a checked checkbox icon (right).

Agora que temos as nossas listas, podemos experimentar com mais um pouco de Ajax e também com alguns outros aspectos do Rails.

Digamos que queremos a possibilidade de ordenar as listas acima por prioridade de ação que queremos executar dentro de cada contexto. Queremos uma maneira visual de fazer isso, bem rápida e fácil. Aqui entra outra implementação Ajax presente no Rails: listas ordenáveis por meio de operações *drag and drop*.

No caso acima, é bem fácil fazer isso. Basta modificar o nosso *partial* mais uma vez. Aproveitamos também para resolver o caso de não termos nenhuma ação completada, deixando somente o grupo visível:

```

<% if actions.any? %>
  <ul id="<% if actions.first.done? %>completed_actions<% else %>context_actions_<%
actions.first.context.id %><% end %>">
    <% actions.each do |action| %>
      <li id="context_action_<%= action.id %>">
        <% unless action.done? %>
          <%= check_box_tag "completed[#{action.id}]", action.id, false, :onclick =>
"#(remote_function(:url => { :action => "mark", :id => action }, :before =>
%(Element.show("indicator")), :complete => %(Element.hide("indicator"))))" %>
        <% end %>
        <%= action.description %><br>
        <span>
          <strong>Project:</strong> <%= action.project.name %>
          <% if action.done? %>
            <br><strong>Context:</strong> <%= action.context.name %>
            <br><strong>Completed At:</strong> <%= action.completed_at.strftime("%m/%d/%y") %>
          <% end %>
        </span>
      </li>
    <% end %>
  </ul>
  <% unless actions.first.done? %>
    <%= sortable_element("context_actions_#{actions.first.context.id}") %>
  <% end %>
<% else %>
  <ul id="completed_actions"></ul>
<% end %>

```

No código acima, queremos somente que as nossas listas de ações não completadas sejam priorizadas, já que, obviamente, não precisamos nos importar mais com ações já terminadas.

Depois de criamos um *id* para cada uma de nossas listas, executamos a chamada ao método `sortable_element`. Esse método é responsável por gerar o código JavaScript necessário para que um elemento HTML representando uma lista possa ser reordenável através de operações *drag and drop*.

Experimente agora clicar sobre um item da lista e movê-lo. Como você pode ver, ao clicar em um elemento a seleção do mesmo se torna livre e ele pode ser arrastada para qualquer outra posição na lista a que pertence. Cada lista também é delimitada e elementos não pode ser trocados em nosso exemplo (embora isso seja possível usando uma ligeira modificação do código).

O resultado de um movimento pode ser visto abaixo:

You are logged in as **Ronaldo**. [Log out.](#)

» Contexts » Projects » Actions » Resources » Users

Next Actions

@Home

- Buy a grammar to help me with my English.
Project: Write a Book
- Sit down and write, every day, at least two hours a day.
Project: Write a Book
- Upgrade my word processor.
Project: Write a Book

@Work

- Plant the tree in the lounge.
Project: Plant a Tree

Completed Actions

Get permission from the powers that be to plant a tree in the lounge.

Project: Plant a Tree
Context: @Work
Completed At: 09/25/06

Done

Obviamente, nosso ordenação ainda não é persistida. Para isso, precisamos de uma alteração em nossos modelos de dados: um campo que marque a posição do elemento na lista. Geramos uma migração para isto:

```
ronaldo@minerva:~/tmp/gtd$ script/generate migration add_priority_to_actions
exists db/migrate
create db/migrate/012_add_priority_to_actions.rb
```

E a editamos:

```

class AddPriorityToActions < ActiveRecord::Migration
  def self.up
    add_column :actions, :position, :integer
  end

  def self.down
    remove_column :actions, :position
  end
end

```

Agora que temos um campo, podemos alterar o método que estamos usando para usá-lo:

```

class Action < ActiveRecord::Base

attr_protected :created_at, :completed_at

def done=(value)
  if ActiveRecord::ConnectionAdapters::Column.value_to_boolean(value)
    self.completed_at = DateTime.now
  else
    self.completed_at = nil
  end
  write_attribute("done", value)
end

belongs_to :context
belongs_to :project
belongs_to :user

validates_presence_of :description

validates_presence_of :context_id
validates_presence_of :project_id

has_and_belongs_to_many :resources, :order => "name"

def self.find_by_status(done, limit = nil)
  find :all,
    :conditions => ["done = ?", done ? 1 : 0],
    :include => [:user, :project, :context],
    :order => "actions.position, actions.description",
    :limit => limit
end

end

```

Isso garante que não tenhamos que nos preocupar com o uso da ordenação quando implementarmos o ação que precisamos no *controller*. Agora, precisamos mudar o nosso *partial* para adicionar uma ação às nossas listas:

```

<% if actions.any? %>
  <ul id=<% if actions.first.done? %>completed_actions<% else %>context_actions<=%
  actions.first.context.id %><% end %>">
    <% actions.each do |action| %>
      <li id="context_action_<=% action.id %>">
        <% unless action.done? %>
          <%= check_box_tag "completed[#{action.id}]", action.id, false, :onclick =>
          "#{remote_function(:url => { :action => "mark", :id => action }, :before =>
          %(Element.show('indicator')), :complete => %(Element.hide('indicator')))}" %>
        <% end %>
        <%= action.description %><br>
        <span>

```

```

<strong>Project:</strong> <%= action.project.name %>
<% if action.done? %>
  <br><strong>Context:</strong> <%= action.context.name %>
  <br><strong>Completed At:</strong> <%= action.completed_at.strftime("%m/%d/%y") %>
<% end %>
</span>
</li>
<% end %>
</ul>
<% unless actions.first.done? %>
<%= sortable_element("context_actions_#{actions.first.context.id}", :url => { :action => "order",
:id => actions.first.context.id }) %>
<% end %>
<% else %>
<ul id="completed_actions"></ul>
<% end %>

```

Embora na ação seja razoavelmente fácil descobrir qual é o elemento que estamos ordenando, passar o *id* diretamente fica bem mais fácil. A implementação seria:

```

class HomeController < ApplicationController

def index
  @actions = Action.find_by_status(false)
  @completed_actions = Action.find_by_status(true, 3)
end

def mark
  @action = Action.find(params[:id])
  @action.update_attribute("done", true)
  @completed_actions = Action.find_by_status(true, 3)
  @remove_context = @action.context.actions.count("done = 0") == 0
end

def order
  ordered_actions = params["context_actions_#{params[:id]}"].collect { |item| item.gsub(/\^0-9/, "") }.to_i
  ordered_actions.each_index do |i|
    action = Action.find(ordered_actions[i]).update_attribute("position", i + 1)
  end
  render :nothing => true
end

end

```

A ação é bem simples. O código JavaScript serializa os dados em um parâmetro cujo nome é o *id* da lista. Recuperamos esse parâmetro, que é meramente um *array* com os elementos em ordem e coletamos esses elementos, removendo tudo que não seja um dígito dos mesmos a fim de deixar somente o *id* de cada um. Em seguida, fazemos um *loop* com esses *id*, recuperando a ação e salvando sua nova posição. Por fim, não retornamos nada já que a ação visual já foi efetuada.

AGINDO COMO UMA LISTA

Embora não tenhamos usado essa funcionalidade aqui, por não termos necessidade, o Rails possui um método que pode fazer com que uma classe haja como uma lista em termos de modelo de dados. Se modificarmos o nosso modelo de ações para o abaixo, teremos isso:

```

class Action < ActiveRecord::Base

  acts_as_list :scope => 'context_id = #{context_id} and done = 0', :column => "position"

  attr_protected :created_at, :completed_at

  def done=(value)
    if ActiveRecord::ConnectionAdapters::Column.value_to_boolean(value)
      self.completed_at = DateTime.now
    else
      self.completed_at = nil
    end
    write_attribute("done", value)
  end

  belongs_to :context
  belongs_to :project
  belongs_to :user

  validates_presence_of :description

  validates_presence_of :context_id
  validates_presence_of :project_id

  has_and_belongs_to_many :resources, :order => "name"

  def self.find_by_status(done, limit = nil)
    find :all,
      :conditions => ["done = ?", done ? 1 : 0],
      :include => [:user, :project, :context],
      :order => "actions.position, actions.description",
      :limit => limit
  end
end

```

A declaração usada no código acima cria uma série de métodos que permitem que a classe se comporte com uma lista quando conjuntos de elementos seus são retornados.

No exemplo, indicamos que a coluna no banco que indica a posição é `position`, que criamos anteriormente. Indicamos também um escopo para a operação da lista. A exemplo do que definimos em Ajax acima, queremos que a lista só seja válida no escopo de ações não completadas dentro de um contexto. Para isso, passamos para o parâmetro `scope` um fragmento de SQL que é usado para especificar os limites da lista. Esse fragmento será usado em todas operações de lista.

Note que definimos esse fragmento com aspas simples. Isso acontece porque estamos usando interpolação e não queremos que ela aconteça imediatamente como aconteceria se estivessemos usando aspas duplas.

O resultado são métodos que podemos usar caso precisamos acessar um grupo de ações com se fosse uma lista. Veja no console como isso funcionaria:

```

ronaldo@minerva:~/tmp/gtd$ script/console
Loading development environment.

>> actions = Action.find_all_by_context_id_and_done(6, false)
=> [#<Action:0xb77c2ae4 @attributes={"context_id"=>"6", "completed_at"=>nil, "project_id"=>"7",
"done"=>"0", "id"=>"5", "description"=>"A simple test action.", "user_id"=>"2", "position"=>"1",

```

```

"created_at"=>"2006-09-24 14:53:57"}>, #<Action:0xb77c2aa8 @attributes={"context_id"=>"6",
"completed_at"=>nil, "project_id"=>"7", "done"=>"0", "id"=>"8", "description"=>"Yet another simple
test action.", "user_id"=>"2", "position"=>"2", "created_at"=>"2006-09-26 18:11:30"}>
>> actions.size
=> 2

>> actions[0].first?
=> true

>> actions[1].last?
=> true

>> actions[1].first?
=> false

>> actions[0].last?
=> false

>> actions[0].move_lower
=> true

>> actions[0].last?
=> true

>> actions[0].first?
=> false

>> actions[0].higher_item
=> #<Action:0xb77e650 @attributes={"context_id"=>"6", "completed_at"=>nil, "project_id"=>"7",
"done"=>"0", "id"=>"8", "description"=>"Yet another simple test action.", "user_id"=>"2",
"position"=>"1", "created_at"=>"2006-09-26 18:11:30"}>

>> actions[0].move_to_top
=> true

```

Como podemos ver pelos exemplos acima, temos métodos bem práticos para a operação da lista. Essa facilidade de criar sub-linguagens declarativas, muito aproveitada pelo Rails, é um dos maiores pontos positivos do Ruby.

Existem outros dois métodos similares que vem com o Rails: `acts_as_tree`, que transforma um modelo em uma árvore aninhada; e `acts_as_nested_set`, uma variação do mesmo tema que, ao contrário da primeira, é muito mais eficiente na busca e muito menos eficiente na atualização da árvore. Existem ainda *plugins* que adicionam comportamentos similares. Como qualquer outra das técnicas que vimos até agora, elas não passam de fragmentos de código que transformam a classe enquanto ela está sendo definida. E nessa simplicidade é que está todo o poder do Ruby e do Rails

Agora que temos a nossa lista funcionando, podemos partir para algo um pouco mais avançado: editar e inserir novas ações via Ajax. Embora inserção e edição seja dois aspectos da mesma moeda, como estamos lidando com Ajax vamos andar um pouco mais devagar.

Para usar a edição, vamos primeiro modificar a nossa *view* principal para exibir um formulário na mesma:

```
<h1>Next Actions</h1>
```

```

<%= render :partial => "edit" %>

<% @actions.group_by(&:context).each do |context, actions| %>
<div class="group" id="<%= context.id %>">

  <h2><%= context.name %></h2>

  <%= render :partial => "actions", :object => actions %>

</div>
<% end %>

<div class="group">

  <h2>Completed Actions</h2>

  <%= render :partial => "actions", :object => @completed_actions %>

</div>

```

É mais fácil usar um *partial* porque podemos ter que modificar o item via Ajax, e isso separa o que precisamos de maneira mais tranquila. O *partial* seria algo assim:

```

<div class="edit" id="edit_form">

<h2>Next Action</h2>

<div class="edit-contents">

  <% form_remote_for :item, @action, :url => { :action => "edit" } do |form| %>

    <p>
      <label for="item_description">Description:</label><br>
      <%= form.text_area "description", :rows => 3 %>
    </p>

    <p>
      <label for="item_context_id">Context:</label><br>
      <%= form.select "context_id", @contexts, :prompt => "--- Choose ---" %>
    </p>

    <p>
      <label for="item_project_id">Project:</label><br>
      <%= form.select "project_id", @projects, :prompt => "--- Choose ---" %>
    </p>

    <p>
      <%= submit_tag "Save" %>
    </p>

  <% end %>

</div>
</div>

```

Note o uso do método `form_remote_for`. Até o momento, usamos o método `start_form_tag` para nossos formulários. Esse é um método simples, que gera somente o *tag* do formulário em si, precisando de `end_form_tag` para fechá-lo.

Existe um outro método, chamado `form_for` (com sua versão remota `form_remote_for`, mostrada acima), que recebe um bloco e nesse bloco é capaz de gerar os campos necessários, usando métodos internos como

demonstrado acima. Além disso, essa versão de geração de formulário também é capaz de usar classes auxiliares para formatar individualmente os campos a serem gerados. A versão remota tem a única diferença de usar Ajax para enviar o campos, de uma maneira similar a `remote_function`.

Para facilitar o nosso tutorial, não vamos inserir em todas as nossas chamadas Ajax o código necessário para mostrar e esconder o indicador de atividade. Incluir o indicador em todas as chamadas é uma boa idéia em qualquer aplicação Ajax, mas para fins de exemplo aqui vamos ignorar essa questão.

E nosso *controller* ficaria assim:

```
class HomeController < ApplicationController

  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
    @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    @projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
  end

  def mark
    @action = Action.find(params[:id])
    @action.update_attribute("done", true)
    @completed_actions = Action.find_by_status(true, 3)
    @remove_context = @action.context.actions.count("done = 0") == 0
  end

  def order
    ordered_actions = params["context_actions_#{params[:id]}"].collect { |item| item.gsub(/\^0-9/, "").to_i }
    ordered_actions.each_index do |i|
      action = Action.find(ordered_actions[i]).update_attribute("position", i + 1)
    end
    render :nothing => true
  end

end
```

Note que, temporariamente, não estamos utilizando nenhum objeto em nosso formulário gerado.

TRANSFORMANDO UM MODELO DE DADOS

Essa é a segunda vez que usamos o código acima para gerar listas de itens a serem usados em elementos `select` em formulários. Com base no princípio DRY, seria interessante mover esse código para um local onde o mesmo pudesse ser reusado. Uma solução é acrescentar um método como o abaixo ao modelo de dados:

```
def as_drop_down(order => "name")
  find(:all, :order => order).collect { |i| [i.name, i.id] }
end
```

Simples, embora obviamente tenha que ser repetida para cada modelo em que tivermos interesse em algo similar. Poderíamos movê-la para um *helper*, com uma pequena modificação e evitar a repetição.

Uma outra solução, melhor ainda, seria criar uma espécie de `acts_as_drop_down`, que poderíamos implementar da seguinte forma:

```
module GTD
  module Acts
    module DropDownList
      def self.append_features(base)
        base.extend(ClassMethods)
      end

      module ClassMethods
        def acts_as_drop_down(options = {})
          class_eval <<-EOF
            def self.as_drop_down
              find(:all, :order => "#{options[:column]}").collect { |i| [i_#{options[:column]}, i.id] }
            end
          EOF
        end
      end
    end
  end
end

ActiveRecord::Base.class_eval do
  include GTD::Acts::DropDown
end
```

No código acima, estamos definindo um módulo `GTD::Acts::DropDown`. Esse módulo usa o método `append_features`, que já explicamos antes, para adicionar novas implementações a uma classe base. Se você reparar as últimas três linhas, verá que estamos incluindo o módulo que acabamos de criar na classe base do `ActiveRecord`, da qual todos nossos modelos derivam.

O nosso método `append_features` não faz mais nada do que usar o método `extend`, presente em todas as classes Ruby, para incluir alguns novos métodos na classe que esta modificando que, como vimos acima, é a classe base do `ActiveRecord`.

Esses métodos estão definidos em um módulo interno, chamado `ClassMethods`. O que esse módulo interno faz, por sua vez, é definir uma extensão chamada `acts_as_drop_down`, que ao ser chamada, transformará nossos modelos da maneira que desejamos aqui, ou seja, introduzirá um método final que podemos usar para gerar nossos dados.

É uma cadeia um pouco complicada de se seguir a princípio, mas estudando o código você verá que ela se torna clara em pouco tempo. Esse método, `acts_as_drop_down`, faz uma coisa bem simples, ele interpreta uma *string* no contexto da classe, que contém a definição final de um método, com base nos parâmetros que passamos. É mais fácil usar uma *string* aqui para evitar as regras de contexto do uso de um bloco.

O método que definimos é o nosso `as_drop_down`, visto anteriormente, mas agora em uma versão genérica. Quando o método `class_eval` roda, ele executa a *string* que lhe foi passada no contexto da classe, gerando um método final que contém o código que precisamos.

O método `acts_as_drop_down` recebe um parâmetro nomeado, *column*, que é a coluna pela qual queremos ordenar e que queremos exibir em nossos formulários. Para facilitar o entendimento, veja o uso do módulo em um modelo de dados:

```
class Context < ActiveRecord::Base
  include GTD::Auditing
  acts_as_drop_down :column => "name"
  attr_protected :user_id
  validates_presence_of :name
  validates_uniqueness_of :name
  validates_presence_of :user_id
  has_many :actions
end
```

Considerando o código acima, são as três linhas finais que nos permitem usar diretamente o método `acts_as_drop_down` sem precisar de algo como fizemos na linha diretamente acima da declaração. Poderíamos usar a mesma técnica para mudar a linha `include GTD::Auditing` para algo como `acts_as_audited`.

No momento em que a classe é definida, o método `acts_as_drop_down` roda, gerando por sua vez um método `as_drop_down` na classe, que podemos usar em nosso *controller* da seguinte maneira:

```
class HomeController < ApplicationController
  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
    @contexts = Context.as_drop_down
    @projects = Project.as_drop_down
  end

  def mark
    @action = Action.find(params[:id])
    @action.update_attribute("done", true)
    @completed_actions = Action.find_by_status(true, 3)
    @remove_context = @action.context.actions.count("done = 0") == 0
  end

  def order
    ordered_actions = params["context_actions_#{params[:id]}"].collect { |item| item.gsub(/[^0-9]/, "") }.to_i
    ordered_actions.each_index do |i|
      action = Action.find(ordered_actions[i]).update_attribute("position", i + 1)
    end
  end
end
```

```

    end
    render :nothing => true
end

end

```

Relembrando a cadeia, temos:

- 1) O módulo GTD::Acts::DropDown é incluído na classe base do *ActiveRecord*, *ActiveRecord::Base*.
- 2) Ao ser incluído, o módulo acima estende a classe base adicionando um método de classe chamado `acts_as_drop_down`, que pode ser invocado para efetuar outra modificação na classe. Com isso conseguimos que a modificação só seja aplicada se o método for realmente invocado, o que fazemos nos modelos de nossa escolha. Isso usa uma propriedade do Ruby que é a modificação dinâmica de uma classe enquanto ela está sendo declarada,
- 3) Caso o método seja invocado, ele executa uma declaração no contexto da classe que gera um método final de classe `as_drop_down`, customizado especialmente para a classe.
- 4) Esse método pode ser usado livremente em *controllers* e *views*, funcionando sem a necessidade de qualquer outra forma de configuração.

Essa técnica de modificação de classes é muito poderosa e poderíamos usá-la e aumentá-la de inúmeras maneiras. De fato, pode-se considerar que basicamente todas as características do Rails são implementadas usando técnicas similares, considerando que essa é uma das maneiras padrão de efetuar mudanças em classes no Ruby.

Isso acontece porque a linguagem não possui, propositadamente, herança múltipla. Assim, a maior parte do que é feito para incrementar uma classe é através de módulos, ou *mixins* na terminologia do Ruby. O próprio método `group_by`, que usamos anteriormente, é um *mixin* adicionado a outro *mixin* que define enumerações.

Obviamente, antes de podemos rodar o código precisamos modificar nosso arquivo de configuração `environment.rb` como visto abaixo e reiniciar o nosso servidor:

```

# Be sure to restart your web server when you modify this file.

# Uncomment below to force Rails into production mode when
# you don't control web/app server and can't set it the proper way
# ENV['RAILS_ENV'] ||= 'production'

# Specifies gem version of Rails to use when vendor/rails is not present
RAILS_GEM_VERSION = '1.1.6'

# Bootstrap the Rails environment, frameworks, and default configuration
require File.join(File.dirname(__FILE__), 'boot')

Rails::Initializer.run do |config|
  # Settings in config/environments/* take precedence those specified here

  # Skip frameworks you're not going to use (only works if using vendor/rails)
  # config.frameworks -= [ :action_web_service, :action_mailer ]

```

```

# Add additional load paths for your own custom dirs
config.load_paths += %W( #{RAILS_ROOT}/extras )

# Force all environments to use the same logger level
# (by default production uses :info, the others :debug)
# config.log_level = :debug

# Use the database for sessions instead of the file system
# (create the session table with 'rake db:sessions:create')
# config.action_controller.session_store = :active_record_store

# Use SQL instead of Active Record's schema dumper when creating the test database.
# This is necessary if your schema can't be completely dumped by the schema dumper,
# like if you have constraints or database-specific column types
# config.active_record.schema_format = :sql

# Activate observers that should always be running
# config.active_record.observers = :cacher, :garbage_collector

# Make Active Record use UTC-base instead of local time
# config.active_record.default_timezone = :utc

# See Rails::Configuration for more options
end

# Add new inflection rules using the following format
# (all these examples are active by default):
# Inflector.inflections do |inflect|
#   inflect.plural /^(ox)$/, '\1en'
#   inflect.singular /^(ox)en/, '\1'
#   inflect.irregular 'person', 'people'
#   inflect.uncountable %w( fish sheep )
# end

# Include your application configuration below
require "user_filter.rb"
require "auditing.rb"
require "acts_as_drop_down.rb"

ActionMailer::Base.server_settings =
{
  :address => "mail.yourdomain.com",
  :domain => "yourdomain.com",
  :user_name => "user@yourdomain.com",
  :password => "*****",
  :authentication => :login
}

```

Essa solução é bem melhor do que as primeiras que pensamos, mas ainda carece de um detalhe: ser portável. Se quisermos usá-la em outros projetos, teremos copiar os arquivos e modificar nossa configuração. A solução ideal seria empacotar o módulo com um *plugin*, que poderia ser simplesmente deixado no diretório apropriado, fazendo com que a mudança estivesse automaticamente disponível para aplicação sem modificações explícitas de configuração.

O método usado também é um pouco limitado no sentido do que o único parâmetro que pode ser informado é o nome de um atributo simples. Em um caso geral, haveria a possibilidade de usar um método qualquer do objeto para texto e outro método para ordenação, condições, limites, e assim por diante. Adicionar isso fica como um exercício para o leitor.

CONTINUANDO COM AJAX

Agora que temos nossas peças, precisamos somente de um pequena alteração em nosso arquivo default.css, adicionando o fragmento de código abaixo:

```
div.group
{
    margin-right: 230px;
}

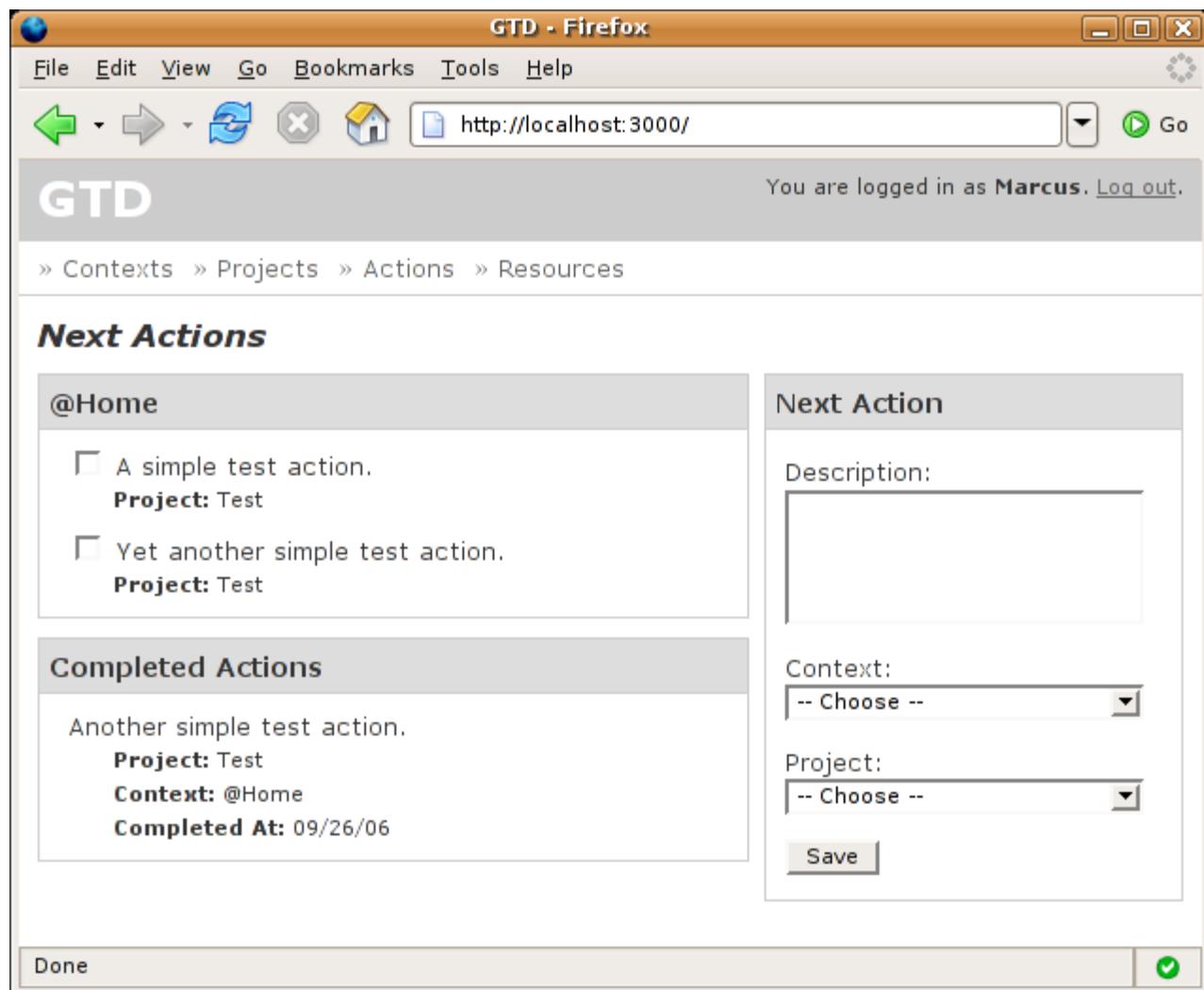
div.edit
{
    border: 1px solid #ccc;
    float: right;
    width: 220px;
}

div.edit-contents
{
    margin: 10px;
}

div.edit h2
{
    padding: 5px;
    margin: 0px;
    border-bottom: 1px solid #ccc;
    background-color: #ddd;
    font-size: 110%;
}

div.edit select,
div.edit textarea
{
    width: 190px;
}
```

O resultado final de tudo o que fizemos até agora, incluindo o desvio acima para mostrar com um módulo pode ser criado, é o seguinte:



Agora que temos o nosso formulário, podemos começar a programação Ajax do mesmo.

Temos que considerar várias situações. Olhando a tela acima, por exemplo, o que acontece quando adicionarmos uma ação em um contexto que não está sendo exibido na página? Como podemos exibir nossas validações? Para resolvêmos isso, vamos ter que fazer alguma modificações razoáveis em nosso código. Mas, para começarmos, vamos simplesmente considerar a existência do contexto na página.

Já temos o nosso formulário remoto. Precisamos agora criar a nossa ação, que salvará o nosso objeto e o exibirá apropriadamente:

```
class HomeController < ApplicationController
  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
    @contexts = Context.as_drop_down
    @projects = Project.as_drop_down
  end
end
```

```

end

def mark
  @action = Action.find(params[:id])
  @action.update_attribute("done", true)
  @completed_actions = Action.find_by_status(true, 3)
  @remove_context = @action.context.actions.count("done = 0") == 0
end

def order
  ordered_actions = params["context_actions_#{params[:id]}"].collect { |item| item.gsub(/[^0-9]/, "") }.to_i
  ordered_actions.each_index do |i|
    action = Action.find(ordered_actions[i]).update_attribute("position", i + 1)
  end
  render :nothing => true
end

def edit
  @action = Action.create(params[:item].merge({ :done => false }))
  unless @action.valid?
    render :nothing => true
  end
end

end

```

O que fazemos na ação é tentar criar e salvar os nossos dados. Se não conseguirmos, temporariamente ignoraremos o erro. No exemplo acima, adicionamos o atributo `done` como falso para que o mesmo receba uma valor não nulo, já que ele não está sendo enviado do formulário.

Agora precisamos de um *template RJS* que nos permita adicionar uma ação recém criada à sua lista de contexto correspondente. Para isso criamos a `view/edit.rjs`:

```

page.insert_html :bottom, "context_actions_#{@action.context.id}", :partial => "item", :object =>
@action
page.visual_effect :highlight, "context_action_#{@action.id}", :duration => 3.5

```

Estamos mais uma vez usando um *partial* para nosso conteúdo, adicionando HTML a um elemento já existente. Como mencionamos acima, vamos lidar depois com o caso de um contexto que não está na página.

Esse *partial* seria algo assim, definido no arquivo `_item.rhtml`:

```

<li id="context_action_<%= item.id %>">
  <% unless item.done? %>
    <%= check_box_tag "completed[#{item.id}]", item.id, false, :onclick => "#{remote_function(:url =>
{ :action => "mark", :id => item }, :before => %(Element.show("indicator")), :complete =>
%(Element.hide("indicator")))}" %>
  <% end %>
  <%= item.description %><br>
  <span>
    <strong>Project:</strong> <%= item.project.name %>
    <% if item.done? %>
      <br><strong>Context:</strong> <%= item.context.name %>
      <br><strong>Completed At:</strong> <%= item.completed_at.strftime("%m/%d/%Y") %>
  
```

```
<% end %>
</span>
</li>
```

O que vamos aqui é nada mais do que tínhamos em nosso outro *partial*, definido no arquivo `_actions.rhtml`. Podemos, consequentemente, mudar esse arquivo para:

```
<% if actions.any? %>
  <ul id="<% if actions.first.done? %>completed_actions<% else %>context_actions_<=%
actions.first.context.id %><% end %>">
    <%= render :partial => "item", :collection => actions %>
  </ul>
  <% unless actions.first.done? %>
    <%= sortable_element("context_actions_#{actions.first.context.id}", :url => { :action => "order",
:id => actions.first.context.id }) %>
    <% end %>
  <% else %>
    <ul id="completed_actions"></ul>
  <% end %>
```

É claro que usar muitos *partials* ocasiona uma perda de performance. Mas não precisamos nos preocupar com isso no momento já que isso seria uma otimização prematura. Um princípio bom do desenvolvimento de aplicações, seja em Rails ou não, é fazer o que seria mais elegante e mais legível primeiro e depois otimizar de acordo com a necessidade.

Em termos de otimização, existem dezenas de opções que poderiam ser consideradas. Mas, quaisquer que sejam só devem ser utilizadas depois da identificação de um problema real para evitar perdas de tempo.

Para a maior parte das aplicações, a velocidade de qualquer página será boa o suficiente porque nem a carga de dados nem a carga de usuários serão altas demais para causar algum problema. E mesmo antes de considerarmos ações complicadas para salvar um ou outro segundo de performance aqui e ali, podem existir soluções mais simples presentes no Rails.

De qualquer forma, aplicações Ajax geralmente balançam um gasto de performance maior por causa de vários *partials* com um tráfego menor de dados na página, por evitarem recarregar os dados completos toda a cada vez que uma mudança é feita.

No caso acima de nossa página, por exemplo, em muitas das ações estaremos tratando com apenas um objeto de dados, e tentar otimizar qualquer coisa relacionada as ações efetuadas em um único objeto seria provavelmente um imenso desperdício de energia.

Voltando à nossa aplicação, o resultado, ao adicionarmos uma ação em um contexto já existente é:

Precisamos agora apenas de refinamentos para melhorar o que já fizemos. A primeira coisa é limpar o formulário em caso de sucesso. Podemos fazer isso da seguinte maneira:

```
page.insert_html :bottom, "context_actions_#{@action.context.id}", :partial => "item", :object =>
@action
page.visual_effect :highlight, "context_action_#{@action.id}", :duration => 3.5
page.replace "edit_form", :partial => "edit"
```

O que requer uma mudança em nosso *controller*:

```
class HomeController < ApplicationController

  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
    @contexts = Context.as_drop_down
    @projects = Project.as_drop_down
  end
end
```

```

end

def mark
  @action = Action.find(params[:id])
  @action.update_attribute("done", true)
  @completed_actions = Action.find_by_status(true, 3)
  @remove_context = @action.context.actions.count("done = 0") == 0
end

def order
  ordered_actions = params["context_actions_#{params[:id]}"].collect { |item| item.gsub(/[^0-9]/, "") }.to_i
  ordered_actions.each_index do |i|
    action = Action.find(ordered_actions[i]).update_attribute("position", i + 1)
  end
  render :nothing => true
end

def edit
  @contexts = Context.as_drop_down
  @projects = Project.as_drop_down
  @action = Action.create(params[:item].merge({ :done => false }))
  unless @action.valid?
    render :nothing => true
  end
end

end

```

No caso acima, como encapsulamos a nossa gerações dos dados necessários de contexto e projeto, não seria uma violação da estratégia MVC mover esses chamadas para dentro da *view*, economizando uma repetição de código.

O resultado seria uma *view* assim:

```

<div class="edit" id="edit_form">

<h2>Next Action</h2>

<div class="edit-contents">

  <% form_remote_for :item, @action, :url => { :action => "edit" } do |form| %>

    <p>
      <label for="item_description">Description:</label><br>
      <%= form.text_area "description", :rows => 3 %>
    </p>

    <p>
      <label for="item_context_id">Context:</label><br>
      <%= form.select "context_id", Context.as_drop_down, :prompt => "-- Choose --" %>
    </p>

    <p>
      <label for="item_project_id">Project:</label><br>
      <%= form.select "project_id", Project.as_drop_down, :prompt => "-- Choose --" %>
    </p>

    <p>
      <%= submit_tag "Save" %>
    </p>

  <% end %>

```

```
</div>  
</div>
```

Com isso, as linhas recém-inseridas poderiam ser removidas do *controller*, sem prejuízo para a manutenção e legibilidade.

Continuando em nossas melhorias, é hora de introduzir validação em nosso formulário. Podemos fazer isso com uma mudança relativamente simples. Primeiro, em nossa *view*:

```
<div class="edit" id="edit_form">  
  <h2>Next Action</h2>  
  <%= error_messages_for "action" %>  
  
  <div class="edit-contents">  
    <% form_remote_for :item, @action, :url => { :action => "edit" } do |form| %>  
  
    <p>  
      <label for="item_description">Description:</label><br>  
      <%= form.text_area "description", :rows => 3 %>  
    </p>  
  
    <p>  
      <label for="item_context_id">Context:</label><br>  
      <%= form.select "context_id", Context.as_drop_down, :prompt => "-- Choose --" %>  
    </p>  
  
    <p>  
      <label for="item_project_id">Project:</label><br>  
      <%= form.select "project_id", Project.as_drop_down, :prompt => "-- Choose --" %>  
    </p>  
  
    <p>  
      <%= submit_tag "Save" %>  
    </p>  
  <% end %>  
  </div>  
</div>
```

Mudando em seguida o nosso *template RSJ*:

```
if @action.valid?  
  page.insert_html :bottom, "context_actions_#{@action.context.id}", :partial => "item", :object =>  
  @action  
  page.visual_effect :highlight, "context_action_#{@action.id}", :duration => 3.5  
  @action = nil  
end  
page.replace "edit_form", :partial => "edit"
```

Só adicionamos um item se o mesmo for válido, e depois renderizamos o nosso *partial* em qualquer circunstância.

A chave aqui é que se tivermos um valor no objeto, o *partial* renderizará um formulário preenchido, caso contrário o formulário virá vazio. Como estamos usando uma variável de instância, zeramos a mesma antes de renderizar o formulário em caso de uma inserção positiva. Agora só precisamos mudar o nosso *controller*:

```
class HomeController < ApplicationController

  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
  end

  def mark
    @action = Action.find(params[:id])
    @action.update_attribute("done", true)
    @completed_actions = Action.find_by_status(true, 3)
    @remove_context = @action.context.actions.count("done = 0") == 0
  end

  def order
    ordered_actions = params["context_actions_#{params[:id]}"].collect { |item| item.gsub(/\^0-9/, "") }.to_i
    ordered_actions.each_index do |i|
      action = Action.find(ordered_actions[i]).update_attribute("position", i + 1)
    end
    render :nothing => true
  end

  def edit
    values = params[:item].merge({ :done => false })
    values["context_id"] = "" if values["context_id"] && values["context_id"].starts_with?("—")
    values["project_id"] = "" if values["project_id"] && values["project_id"].starts_with?("—")
    @action = Action.create(values)
  end
end
```

O código acima se deve a uma pequena característica da biblioteca Prototype usada para o Rails. Ao serializar formulários para envio via JavaScript, no caso de elementos do tipo *select*, se o item selecionado não tiver um valor associado a biblioteca envia o texto, ao contrário do que navegadores fazem normalmente. Isso é o que tratamos no código acima.

No nosso formulário, isso acontece porque os elementos *select* que estamos usando contém um valor vazio para o primeiro elemento, que é gerado pelo parâmetro *prompt* para nos dar uma usabilidade maior.

Uma solução para o problema seria modificar a biblioteca para não ter esse comportamento por padrão. O inconveniente seria ter que manter esse *patch* a cada atualização da mesma. Uma outra solução seria criar um filtro para automaticamente limpar os nossos parâmetros se for o caso. E uma terceira solução seria não usar o parâmetro *prompt*, usando *include_blank* ao invés disso, com uma pequena perda de usabilidade.

O resultado, com a validação, é o seguinte:

Agora precisamos adicionar o caso em que um contexto que não está sendo exibido na página, por não possuir nenhuma ação não completada, recebe uma ação.

Mudamos a nossa view primária:

```
<h1>Next Actions</h1>
<%= render :partial => "edit" %>
<div id="contexts">
```

```

<% @actions.group_by(&:context).each do |context, actions| %>
<%= render :partial => "context", :object => context, :locals => { :actions => actions } %>
<% end %>
</div>

<div class="group">
  <h2>Completed Actions</h2>
  <%= render :partial => "actions", :object => @completed_actions %>
</div>

```

E, como consequência, criamos uma novo *partial*, definido no arquivo `_context.rhtml`:

```

<div class="group" id="context_<%= context.id %>">
  <h2><%= context.name %></h2>
  <%= render :partial => "actions", :object => actions %>
</div>

```

Por fim, alteramos o nosso *template* RSJ:

```

if @action.valid?
  page << "if (typeof($('context_#{@action.context.id}')) == 'undefined') {" 
  page.insert_html :top, "contexts", :partial => "context", :object => @action.context, :locals =>
  { :actions => @action.context.actions }
  page << "} else {" 
  page.insert_html :bottom, "context_actions_#{@action.context.id}", :partial => "item", :object =>
  @action
  page << "}
  page.visual_effect :highlight, "context_action_#{@action.id}", :duration => 3.5
  @action = nil
end
page.replace "edit_form", :partial => "edit"

```

No exemplo acima, precisamos saber se o contexto já existe na página. Para isso, usamos um pouco de JavaScript direto, verificando se há um objeto com o *id* que precisamos na página. Caso exista, simplesmente adicionamos a ação. Caso contrário, renderizamos o contexto completo e o inserimos no topo da lista geral de contextos. O resultado final é o que queremos, com dois problemas simples:

O primeiro é que tanto as ações como contextos inseridos estão potencialmente fora de ordenação de acordo com os critérios que estabelecemos já que estamos inserindo o resultado de nossas chamadas em locais fixos. A solução para isso seria renderizarmos as seções completas que precisamos, mesmo que isso seja um pouco mais caro em termos de performance. Se a ordem for muito importante, essa é uma solução válida.

O segundo problema é termos que forçar a variável `@action` a assumir um valor nulo antes de exibirmos o formulário para o caso de querermos um formulário vazio. O melhor seria usar a própria convenção do Rails e usar uma variável definida para o próprio *partial*. O problema, nesse caso, é que estamos usando

`error_messages_for`, que espera uma variável de instância. Para fins do nosso tutorial isso não tem muita importância. Mas para um aplicação real, seria interessante uma redefinição desse método.

O resultado da adição de uma ação em um contexto não existente na página é o seguinte:

The screenshot shows a Firefox browser window titled "GTD - Firefox". The address bar displays "http://localhost:3000/". The main content area of the GTD application shows the following sections:

- Next Actions**:
 - @Work**:
 - A test action added via Ajax.
Project: Test
 - @Home**:
 - A simple test action.
Project: Test
 - Yet another simple test action.
Project: Test
- Completed Actions**:
 - Another simple test action.
Project: Test
Context: @Home
Completed At: 09/26/06

To the right, a modal dialog titled "Next Action" is open, containing fields for "Description", "Context" (with a dropdown menu showing "-- Choose --"), "Project" (with a dropdown menu showing "-- Choose --"), and a "Save" button.

Se você tentou usar a lista para reordenar um elemento depois da adição de um item, reparou que a mesma para de funcionar. Esse é o comportamento normal de uma lista quando um novo elemento é adicionado à mesma. Para corrigir isso, basta reinicializar a lista depois de uma modificação na mesma, corrigindo o *template RJS* para edição:

```

if @action.valid?
  page << "if (typeof('${@action.context.id}') == 'undefined') {"
  page.insert_html :top, "contexts", :partial => "context", :object => @action.context, :locals =>
{ :actions => @action.context.actions }
  page << "} else {"
  page.insert_html :bottom, "context_actions_#{@action.context.id}", :partial => "item", :object =>
@action
  page << "}"
  page.visual_effect :highlight, "context_action_#{@action.id}", :duration => 3.5
  page.sortable "context_actions_#{@action.context.id}", :url => { :action => "order", :id =>
@action.context.id }
  @action = nil
end
page.replace "edit_form", :partial => "edit"

```

E uma última melhoria que podemos fazer antes de partirmos para o nosso próximo tópico é ter alguma forma de cancelar a edição quando o usuário desejar. Para isso, precisamos de modificar o nosso *partial* de edição, em `_edit.rhtml`:

```

<div class="edit" id="edit_form">

<h2>Next Action</h2>

<%= error_messages_for "action" %>

<div class="edit-contents">

  <% form_remote_for :item, @action, :url => { :action => "edit" } do |form| %>

    <p>
      <label for="item_description">Description:</label><br>
      <%= form.text_area "description", :rows => 3 %>
    </p>

    <p>
      <label for="item_context_id">Context:</label><br>
      <%= form.select "context_id", Context.as_drop_down, :prompt => "-- Choose --" %>
    </p>

    <p>
      <label for="item_project_id">Project:</label><br>
      <%= form.select "project_id", Project.as_drop_down, :prompt => "-- Choose --" %>
    </p>

    <p>
      <%= submit_tag "Save" %> or
      <%= link_to_remote "Cancel", :url => { :action => "cancel" } %>
    </p>

  <% end %>

</div>

</div>

```

E depois disso, adicionar uma ação ao nosso *controller*:

```

class HomeController < ApplicationController

```

```

def index
  @actions = Action.find_by_status(false)
  @completed_actions = Action.find_by_status(true, 3)
end

def mark
  @action = Action.find(params[:id])
  @action.update_attribute("done", true)
  @completed_actions = Action.find_by_status(true, 3)
  @remove_context = @action.context.actions.count("done = 0") == 0
end

def order
  ordered_actions = params["context_actions_#{params[:id]}"].collect { |item| item.gsub(/[^0-9]/, "").to_i }
  ordered_actions.each_index do |i|
    action = Action.find(ordered_actions[i]).update_attribute("position", i + 1)
  end
  render :nothing => true
end

def edit
  values = params[:item].merge({ :done => false })
  values["context_id"] = "" if values["context_id"] && values["context_id"].starts_with?("--")
  values["project_id"] = "" if values["project_id"] && values["project_id"].starts_with?("--")
  @action = Action.create(values)
end

def cancel
end

end

```

Finalizando com um novo *template* RJS para a ação, gerando uma *view* chamada `cancel.rjs`:

```
page.replace "edit_form", :partial => "edit"
```

Uma outra forma de fazer o cancelamento seria renderizar diretamente o *partial*, pulando o *template* RJS, que não existiria.

Para isso, nosso formulário de edição ficaria um pouco diferente:

```

<div id="edit_form_container">

<div class="edit" id="edit_form">

<h2>Next Action</h2>

<%= error_messages_for "action" %>

<div class="edit-contents">

  <% form_remote_for :item, @action, :html => { :action => url_for(:action => "save", :id => @action) }, :url => { :action => "save", :id => @action } do |form| %>

    <p>
      <label for="item_description">Description:</label><br>
      <%= form.text_area "description", :rows => 3 %>
    </p>

```

```

<% if !@action || @action.new_record? %>

<p>
  <label for="item_context_id">Context:</label><br>
  <%= form.select "context_id", Context.as_drop_down, :prompt => "-- Choose --" %>
</p>

<p>
  <label for="item_project_id">Project:</label><br>
  <%= form.select "project_id", Project.as_drop_down, :prompt => "-- Choose --" %>
</p>

<% end %>

<p>
  <%= submit_tag "Save" %> or
  <%= link_to_remote "Cancel", :update => "edit_form_container", :url => { :action => "cancel" } %>
</p>

<% end %>

</div>
</div>
</div>

```

Também teríamos uma ligeira modificação em nosso *controller*:

```

class HomeController < ApplicationController

  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
  end

  def mark
    @action = Action.find(params[:id]).gsub(/[^0-9]/, "")
    @action.update_attribute("done", true)
    @completed_actions = Action.find_by_status(true, 3)
    @remove_context = @action.context.actions.count("done = 0") == 0
  end

  def order
    ordered_actions = params["context_actions_#{params[:id]}"].collect { |item| item.gsub(/[^0-9]/, "").to_i }
    ordered_actions.each_index do |i|
      action = Action.find(ordered_actions[i]).update_attribute("position", i + 1)
    end
    render :nothing => true
  end

  def edit
    @action = params[:id] ? Action.find(params[:id]) : Action.new
  end

  def save
    values = params[:item].merge({ :done => false })
    values["context_id"] = "" if values["context_id"] && values["context_id"].starts_with?("--")
    values["project_id"] = "" if values["project_id"] && values["project_id"].starts_with?("--")
    @action = params[:id] ?
      Action.update(params[:id], values) :
      Action.create(values)
    respond_to do |wants|
      wants.js
      wants.html do
        if @action.valid?

```

```

    redirect_to :action => "index"
else
  index
  render :action => "index"
end
end
end

def cancel
  render :partial => "edit"
end

def delete
  @action = Action.find(params[:id])
  @action.destroy
  @remove_context = @action.context.actions.count("done = 0") == 0
end

end

```

A título de observação, note que os exemplos seguintes **desfazem** essa modificação.

A maneira usada acima faz uso de uma característica das invocações Ajax realizadas no Rails que é a possibilidade de, ao invés de retornar JavaScript, retornar um fragmento de HTML e atualizar diretamente a página com o mesmo. O parâmetro `update`, no exemplo acima, define o elemento HTML que receberá o conteúdo. Precisamos de um elemento superior porque a substituição é feita somente no conteúdo externo. O parâmetro `update` pode também discriminar entre sucesso e falha usando a seguinte configuração:

```
:update => { :success => "id1", :failure => "id2" }
```

No caso acima, se a ação retornar um código HTTP de sucesso, o elemento com o *id* especificado pelo sub-parâmetro `success` é que receberá o conteúdo retornado; caso contrário, o elemento cujo *id* é especificado pelo outro parâmetro é que será atualizado.

As chamadas remotas no Rails possuem várias outras características que gastaríamos muito tempo para explorar aqui e cujas descrições podem ser encontradas na documentação e em outros tutoriais mais específicos. Entre essas características incluem-se a passagem de parâmetros customizados para as ações no servidor, acompanhamento do processo completo de requisição, tratamento de confirmações e uma série de outras possibilidades que nos permitem refinar bastante a invocação do método remoto.

Isso termina nossas melhorias.

Depois de termos a inserção funcionando, podemos trabalhar com a edição e remoção de ações existentes.

Antes de partirmos para isso porém, uma dica: existe um *plugin* para o Rails, chamado Unobtrusive JavaScript (<http://www.ujs4rails.com>) que pode ajudar bastante na hora de gerar código JavaScript interpolado com uma página para invocar *links*, funções e formulários remotos.

Vamos começar com a exclusão de um item, que segue uma linha muito similar à de marcar o mesmo como completo. Modificamos o nosso *partial* que gerar um item para incluir um *link* remoto para a ação:

```
<li id="context_action_<%= item.id %>">
  <% unless item.done? %>
    <%= check_box_tag "completed[#{item.id}]", item.id, false, :onclick => "#{remote_function(:url =>
{ :action => "mark", :id => item }, :before => %(Element.show("indicator")), :complete =>
%(Element.hide("indicator")))}" %>
  <% end %>
  <%= item.description %>
  <% unless item.done? %>
    <%= link_to_remote "[D]", { :url => { :action => "delete", :id => item }, :confirm => "Are you
sure?", :class => "action-link" } %>
  <% end %>
  <br>
  <span>
    <strong>Project:</strong> <%= item.project.name %>
    <% if item.done? %>
      <br><strong>Context:</strong> <%= item.context.name %>
      <br><strong>Completed At:</strong> <%= item.completed_at.strftime("%m/%d/%y") %>
    <% end %>
  </span>
</li>
```

Acompanhado do seguinte fragmento CSS a ser adicionado em nosso arquivo default.css:

```
a.action-link
{
  text-decoration: none;
  font-size: smaller;
  border-bottom: 1px dashed #999;
  color: #666;
}
```

Agora que temos a nossa chamada, podemos implementar a nossa ação:

```
class HomeController < ApplicationController

  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
  end

  def mark
    @action = Action.find(params[:id])
    @action.update_attribute("done", true)
    @completed_actions = Action.find_by_status(true, 3)
    @remove_context = @action.context.actions.count("done = 0") == 0
  end

  def order
    ordered_actions = params["context_actions_#{params[:id]}"].collect { |item| item.gsub(/[^0-9]/,
"").to_i }
    ordered_actions.each_index do |i|
      action = Action.find(ordered_actions[i]).update_attribute("position", i + 1)
    end
    render :nothing => true
  end

  def edit
    values = params[:item].merge({ :done => false })
  end
end
```

```

values["context_id"] = "" if values["context_id"] && values["context_id"].starts_with?("--")
values["project_id"] = "" if values["project_id"] && values["project_id"].starts_with?("--")
@action = Action.create(values)
end

def cancel
end

def delete
  @action = Action.find(params[:id])
  @action.destroy
  @remove_context = @action.context.actions.count("done = 0") == 0
end

end

```

E, finalmente, nosso *template RJS* que constitui a *view* delete.rjs:

```

page.visual_effect :fade, "context_action_#{@action.id}"
page.delay(5) do
  page.remove "context_action_#{@action.id}"
end

if @remove_context
  page.remove "context_#{@action.context.id}"
end

```

Introduzimos um pequeno *delay* na execução da remoção para que o efeito de desaparecimento possa rodar primeiro.

Isso completa a nossa remoção.

Agora podemos passar para a nossa edição. Para a mesma, vamos separar a ação que faz a edição da ação que salva realmente os dados.

Mudamos inicialmente o nosso *controller*:

```

class HomeController < ApplicationController

  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
  end

  def mark
    @action = Action.find(params[:id])
    @action.update_attribute("done", true)
    @completed_actions = Action.find_by_status(true, 3)
    @remove_context = @action.context.actions.count("done = 0") == 0
  end

  def order
    ordered_actions = params["context_actions_#{params[:id]}"].collect { |item| item.gsub(/[^0-9]/, "") }.to_i
    ordered_actions.each_index do |i|
      action = Action.find(ordered_actions[i]).update_attribute("position", i + 1)
    end
    render :nothing => true
  end

```

```

end

def edit
  @action = params[:id] ? Action.find(params[:id]) : Action.new
end

def save
  values = params[:item].merge({ :done => false })
  values["context_id"] = "" if values["context_id"] && values["context_id"].starts_with?("--")
  values["project_id"] = "" if values["project_id"] && values["project_id"].starts_with?("--")
  @action = params[:id] ?
    Action.update(params[:id], values) :
    Action.create(values)
end

def cancel
end

def delete
  @action = Action.find(params[:id])
  @action.destroy
  @remove_context = @action.context.actions.count("done = 0") == 0
end

end

```

Depois disso, renomeamos a nossa *view* atualmente chamada `edit.rjs` para `save.rjs`, que termina com o seguinte conteúdo:

```

if @action.valid?
  page << "if (typeof($('context_#{@action.context.id}')) == 'undefined') {}"
  page.insert_html :top, "contexts", :partial => "context", :object => @action.context, :locals =>
{ :actions => @action.context.actions }
  page << "} else {"
  if params[:id]
    page.replace "context_action_#{@action.id}", :partial => "item", :object => @action
  else
    page.insert_html :bottom, "context_actions_#{@action.context.id}", :partial => "item", :object =>
@action
  end
  page << "}"
  page.visual_effect :highlight, "context_action_#{@action.id}", :duration => 3.5
  page.sortable "context_actions_#{@action.context.id}", :url => { :action => "order", :id =>
@action.context.id }
  @action = nil
end
page.replace "edit_form", :partial => "edit"

```

Enquanto nossa *view* `edit.rjs` agora contém somente isso:

```
page.replace "edit_form", :partial => "edit"
```

Depois, editamos o nosso *partial* que gera um item, que é o arquivo `_item.rhtml`:

```

<li id="context_action_<%= item.id %>">
  <% unless item.done? %>
    <%= check_box_tag "completed[#{item.id}]", item.id, false, :onclick => "#{remote_function(:url =>
{ :action => "mark", :id => item }, :before => %(Element.show("indicator")), :complete =>
%(Element.hide("indicator")))}" %>

```

```

<% end %>
<%= item.description %>
<% unless item.done? %>
<%= link_to_remote "[E]", { :url => { :action => "edit", :id => item } }, { :class => "action-link" } %>
<%= link_to_remote "[D]", { :url => { :action => "delete", :id => item }, :confirm => "Are you
sure?" }, { :class => "action-link" } %>
<% end %>
<br>
<span>
  <strong>Project:</strong> <%= item.project.name %>
  <% if item.done? %>
    <br><strong>Context:</strong> <%= item.context.name %>
    <br><strong>Completed At:</strong> <%= item.completed_at.strftime("%m/%d/%y") %>
  <% end %>
</span>
</li>

```

Finalmente, editamos o nosso *partial* responsável pela geração do formulário de edição:

```

<div class="edit" id="edit_form">

<h2>Next Action</h2>

<%= error_messages_for "action" %>

<div class="edit-contents">

  <% form_remote_for :item, @action, :url => { :action => "save", :id => @action } do |form| %>

    <p>
      <label for="item_description">Description:</label><br>
      <%= form.text_area "description", :rows => 3 %>
    </p>

    <% if !@action || @action.new_record? %>

      <p>
        <label for="item_context_id">Context:</label><br>
        <%= form.select "context_id", Context.as_drop_down, :prompt => "-- Choose --" %>
      </p>

      <p>
        <label for="item_project_id">Project:</label><br>
        <%= form.select "project_id", Project.as_drop_down, :prompt => "-- Choose --" %>
      </p>

    <% end %>

      <p>
        <%= submit_tag "Save" %> or <%= link_to_remote "Cancel", :url => { :action => "cancel" } %>
      </p>

    <% end %>

  </div>
</div>

```

Quando estamos editando uma ação, não queremos mudar o seu contexto ou projeto. Para isso, bloqueamos esses campos em nosso exemplo.

Obviamente, poderíamos querer editá-los também em uma outra situação, caso qual teríamos que

considerar a possibilidade de mover itens de um local para o outro, o que não seria muito complicado. Fazer isso fica como um exercício para o leitor, considerando que todas as peças necessárias estão disponíveis.

O resultado de uma edição com base no código acima seria:

The screenshot shows a Firefox browser window titled "GTD - Firefox" displaying a GTD application. The URL in the address bar is "http://localhost:3000/". The page header says "You are logged in as Marcus. [Log out](#)". The main navigation menu includes "File", "Edit", "View", "Go", "Bookmarks", "Tools", and "Help". Below the menu, there are standard browser navigation buttons (Back, Forward, Stop, Home) and a search bar.

The application interface includes the following sections:

- Next Actions**: A list of actions categorized by context:
 - @Home**:
 - Yet another simple test action. [\[E\]](#) [\[D\]](#)
Project: Test
 - Another simple test action. [\[E\]](#) [\[D\]](#)
Project: Test
 - @Work**:
 - Yet another test action added via Ajax. [\[E\]](#) [\[D\]](#)
Project: Test
- Completed Actions**: A list of completed actions:
 - A simple test action.
Project: Test
Context: @Home
Completed At: 09/29/06

On the right side of the screen, there is a modal dialog box titled "Next Action" with the following fields:

- Description: Yet another test action added via Ajax.
- Buttons: "Save" and "Cancel".

At the bottom of the application window, there is a "Done" button with a checkmark icon.

Se quisermos mais uma modificação Ajax em nossa página, poderíamos acrescentar a possibilidade de que um usuário arraste uma ação para a lista de ações completadas.

Para isso, precisaríamos, inicialmente, de modificar duas de nossas *views*. Primeiramente, a *view* que gera um item para acrescentar o fato que o mesmo pode ser arrastado.

Ela ficaria assim:

```
<li id="context_action_<%= item.id %>">
  <% unless item.done? %>
    <%= check_box_tag "completed[#{item.id}]", item.id, false, :onclick => "#{remote_function(:url =>
{ :action => "mark", :id => item }, :before => %(Element.show("indicator")), :complete =>
%(Element.hide("indicator")))}" %>
  <% end %>
  <%= item.description %>
  <% unless item.done? %>
    <%= link_to_remote "[E]", { :url => { :action => "edit", :id => item } }, { :class => "action-link" } %>
    <%= link_to_remote "[D]", { :url => { :action => "delete", :id => item }, :confirm => "Are you
sure?" }, { :class => "action-link" } %>
  <% end %>
  <br>
  <span>
    <strong>Project:</strong> <%= item.project.name %>
    <% if item.done? %>
      <br><strong>Context:</strong> <%= item.context.name %>
      <br><strong>Completed At:</strong> <%= item.completed_at.strftime("%m/%d/%y") %>
    <% end %>
  </span>
</li>
<%= draggable_element "context_action_#{item.id}" unless item.done? %>
```

Em seguida, modificamos a nossa *view* primária:

```
<h1>Next Actions</h1>

<%= render :partial => "edit" %>

<div id="contexts">
  <% @actions.group_by(&:context).each do |context, actions| %>
    <%= render :partial => "context", :object => context, :locals => { :actions => actions } %>
  <% end %>
</div>

<div class="group" id="completed_actions_container">
  <h2>Completed Actions</h2>
  <%= render :partial => "actions", :object => @completed_actions %>
  <%= drop_receiving_element "completed_actions_container", :url => { :action => "mark" } %>
</div>
```

E finalmente, modificando o nosso *controller* para receber marcar itens de acordo com a nova situação:

```
class HomeController < ApplicationController

  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
  end

  def mark
    @action = Action.find(params[:id].gsub(/[^\w]/, ""))
    @action.update_attribute("done", true)
    @completed_actions = Action.find_by_status(true, 3)
    @remove_context = @action.context.actions.count("done = 0") == 0
  end
end
```

```

def order
  ordered_actions = params["context_actions_#{params[:id]}"].collect { |item| item.gsub(/[^0-9]/, "") }.to_i
  ordered_actions.each_index do |i|
    action = Action.find(ordered_actions[i]).update_attribute("position", i + 1)
  end
  render :nothing => true
end

def edit
  @action = params[:id] ? Action.find(params[:id]) : Action.new
end

def save
  values = params[:item].merge({ :done => false })
  values["context_id"] = "" if values["context_id"] && values["context_id"].starts_with?("--")
  values["project_id"] = "" if values["project_id"] && values["project_id"].starts_with?("--")
  @action = params[:id] ?
    Action.update(params[:id], values) :
    Action.create(values)
end

def cancel
end

def delete
  @action = Action.find(params[:id])
  @action.destroy
  @remove_context = @action.context.actions.count("done = 0") == 0
end

end

```

Essa modificação é necessária por causa da maneira com os parâmetros são enviados para o *controller* depois de uma operação *drag and drop*. Como você deve ter notado, a ordenação da lista continua funcionando.

DEGRADAÇÃO EM AJAX

Uma coisa importante quando lidamos com Ajax é não deixar que tudo aconteça somente por meio do JavaScript, já que existem situações em que um navegador não suporta a tecnologia ou está com a mesma desabilitado. Para lidar com isso, o Rails possui maneiras de gerar uma degradação de funcionalidade que permite o uso de uma página mesmo com o JavaScript sem funcionar. Tomando como exemplo o nosso formulário de inserção de ações, podemos fazer um rápido teste disso.

Primeiro, modificamos o nosso *partial* de edição:

```

<div class="edit" id="edit_form">

<h2>Next Action</h2>

<%= error_messages_for "action" %>

<div class="edit-contents">

  <% form_remote_for :item, @action, :html => { :action => url_for(:action => "save", :id => @action) } , :url => { :action => "save", :id => @action } do |form| %>

```

```

<p>
  <label for="item_description">Description:</label><br>
  <%= form.text_area "description", :rows => 3 %>
</p>

<% if !@action || @action.new_record? %>

<p>
  <label for="item_context_id">Context:</label><br>
  <%= form.select "context_id", Context.as_drop_down, :prompt => "-- Choose --" %>
</p>

<p>
  <label for="item_project_id">Project:</label><br>
  <%= form.select "project_id", Project.as_drop_down, :prompt => "-- Choose --" %>
</p>

<% end %>

<p>
  <%= submit_tag "Save" %> or <%= link_to_remote "Cancel", :url => { :action => "cancel" } %>
</p>

<% end %>

</div>
</div>

```

O parâmetro *html* do método `form_remote_for` permite especificarmos uma alternativa em HTML comum para o formulário. Agora precisamos de uma modificação em nosso *controller*:

```

class HomeController < ApplicationController

  def index
    @actions = Action.find_by_status(false)
    @completed_actions = Action.find_by_status(true, 3)
  end

  def mark
    @action = Action.find(params[:id].gsub(/[^0-9]/, ""))
    @action.update_attribute("done", true)
    @completed_actions = Action.find_by_status(true, 3)
    @remove_context = @action.context.actions.count("done = 0") == 0
  end

  def order
    ordered_actions = params["context_actions_#{params[:id]}"].collect { |item| item.gsub(/[^0-9]/, "").to_i }
    ordered_actions.each_index do |i|
      action = Action.find(ordered_actions[i]).update_attribute("position", i + 1)
    end
    render :nothing => true
  end

  def edit
    @action = params[:id] ? Action.find(params[:id]) : Action.new
  end

  def save
    values = params[:item].merge({ :done => false })
    values["context_id"] = "" if values["context_id"] && values["context_id"].starts_with?("--")
    values["project_id"] = "" if values["project_id"] && values["project_id"].starts_with?("--")
    @action = params[:id] ?
      Action.update(params[:id], values) :

```

```

Action.create(values)
respond_to do |wants|
  wants.js
  wants.html do
    if @action.valid?
      redirect_to :action => "index"
    else
      index
      render :action => "index"
    end
  end
end

def cancel
end

def delete
  @action = Action.find(params[:id])
  @action.destroy
  @remove_context = @action.context.actions.count("done = 0") == 0
end

end

```

O método `respond_to` é uma adição do Rails 1.1 que permite que uma aplicação reaja de formas diferentes de acordo com o tipo de resposta pedido. No caso acima, se o tipo de resposta pedido é JavaScript, o retorno é deixado como está (primeira linha dentro do bloco); se, ao contrário, HTML é pedido, renderizamos a página novamente. Com o código que já inserimos no *partial* e com o código já existente nessa ação, o objeto com que estivermos trabalhando será salvo tranquilamente. E isso, com apenas um pouco mais de código.

Assim, reagir apropriadamente de acordo com a situação é possível. É claro que a dificuldade varia de caso para caso. No nosso exemplo, marcas ações com completadas seria um pouco mais difícil, mas longe de impossível. Modificar todas as ações de nossa página para funcionar sem Ajax fica como um exercício para o leitor.

Com isso terminamos a nossa visão rápida do Ajax em nosso tutorial. Não vimos tudo o que é possível fazer: por exemplo, *uploads*, inclusive já foi mencionado anteriormente, são um pouco problemáticos de se fazer usando Ajax.

Ajax não é uma solução mágica para os seus problemas de usabilidade. É uma ferramenta muito poderosa para aumentar a capacidade e flexibilidade de suas aplicações, mas, como qualquer outra ferramenta, quem faz a diferença é o desenvolvedor. Pensando nisso, a coisa mais importante em toda a nossa parte de Ajax está justamente no nosso exemplo final de como fazer com que a página continue a funcionar mesmo na ausência de JavaScript. Isso porque acessibilidade é mais importante do que qualquer outra coisa em uma aplicação.

Avançando um pouco mais

Já progredimos bastante em nossa aplicação, cobrindo muito do que é o dia-a-dia do desenvolvimento de um

aplicação Rails. Nesta seção vamos cobrir alguns tópicos que nos permitem extrair ainda mais funcionalidade de nossas aplicações.

Em especial, uma das seções seguintes trata de modificações que permite um uso melhor do Rails em outros idiomas, que provavelmente será o seu caso, desenvolvendo aplicações em português, com mensagens traduzidas, acentos e toda a parafernália necessária para uma aplicação localizada.

Mas, antes de vermos isso, vamos considerar alguns outros assuntos:

ROTEAMENTO AVANÇADO

Até o momento, usamos somente o roteamento padrão fornecido pelo Rails, com uma pequena exceção que é a página inicial da aplicação, redirecionada anteriormente para o *controller* home. Em alguns casos, porém, você terá necessidade de usar URLs especialmente configuradas para a sua aplicação—tanto por questões de usabilidade e legibilidade como por questões de facilidade de desenvolvimento.

No arquivo `routes.rb`, que vimos anteriormente, é possível definir roteamentos diferentes que serão aplicados de acordo com critérios de identificação definidos por você.

Suponhamos, por exemplo, que queremos ver as nossas ações separadas por dia, ao invés da página atual que temos, que simplesmente exibe uma listagem. Podemos utilizar um roteamento customizado para lidar com isso, modificando o arquivo anteriormente mencionado da seguinte forma:

```
ActionController::Routing::Routes.draw do |map|
  # The priority is based upon order of creation: first created -> highest priority.

  # Sample of regular route:
  # map.connect "products/:id", :controller => "catalog", :action => "view"
  # Keep in mind you can assign values other than :controller and :action

  # Sample of named route:
  # map.purchase "products/:id/purchase", :controller => "catalog", :action => "purchase"
  # This route can be invoked with purchase_url(:id => product.id)

  # You can have the root of your site routed by hooking up ""
  # -- just remember to delete public/index.html.
  map.connect "", :controller => "home"

  # Allow downloading Web Service WSDL as a file with an extension
  # instead of a file named "wsdl"
  map.connect ":controller/service.wsdl", :action => "wsdl"

  map.connect ":year/:month/:day", :controller => "actions", :action => "by_date",
    :requirements => {
      :year => /\d{4}/,
      :month => /\d{1, 2}/,
      :day => /\d{1, 2}/
    }

  # Install the default route as the lowest priority.
  map.connect ":controller/:action/:id"
end
```

O roteamento que estabelecemos acima indica que esperamos três itens como a composição de nossa URL. Quando esse roteamento for acionado, esses itens serão respectivamente enviados para parâmetros chamados `year`, `month` e `day`. Esses parâmetros serão passados para a ação `by_date` do `controller` cujo nome é `actions`. O último parâmetro do roteamento indica validações que precisam ser efetuadas antes que aquele roteamento seja invocado. No caso acima, definimos que esperamos o ano com quatro dígitos e mês e dia com um ou dois.

Agora podemos definir a nossa ação, modificando o `controller` existente:

```
class ActionsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @action_pages, @actions = paginate :actions, :include => [:project, :context], :per_page => 5,
    :order => "actions.description"
    render :action => "list"
  end

  def by_date
    @action_pages, @actions = paginate :actions, :conditions => ["actions.created_at = ?",
    Date.new(params[:year].to_i, params[:month].to_i, params[:day].to_i)], :include => [:project,
    :context], :per_page => 5, :order => "actions.description"
    render :action => "list"
  end

  def edit
    @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    @projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    if request.get?
      if params[:id]
        @item = Action.find(params[:id])
      else
        @item = Action.new
      end
    elsif request.post?
      @item = params[:id] ?
        Action.update(params[:id], params[:item]) :
        Action.create(params[:item])
      if @item.valid?
        flash[:notice] = "The action was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def resources
    @action = Action.find(params[:id])
    @resources = []
    if request.post? && !params[:keywords].blank?
      @resources = Resource.find(:all, :conditions => ['name like ? or filename like ?',
      "%#{params[:keywords]}%", "%#{params[:keywords]}%"])
    end
  end

  def add_resources
    @action = Action.find(params[:id])
    existing_resource_ids = @action.resources.collect { |resource| resource.id }
    new_resource_ids = params[:resources].reject { |id| existing_resource_ids.include?(id.to_i) }
    @action.resources << Resource.find(new_resource_ids)
    redirect_to :action => "resources", :id => @action.id
  end

```

```

end

def delete_resource
  @action = Action.find(params[:id])
  @action.resources.delete(Resource.find(params[:resource_id]))
  redirect_to :action => "resources", :id => @action.id
end

def delete
  Action.find(params[:id]).destroy
  flash[:notice] = "The action was successfully deleted"
  redirect_to :action => "list"
end
end

```

A ação criada acima faz uma busca baseada nos parâmetros que passamos, como você pode ver pelo resultado abaixo (note a URL):

Description	Completed	When	Project	Context	Actions
Another simple test action.	No	-	Test	@Home	Edit or Delete or Edit Resources
Yet another simple test action.	Yes	10/01/06	Test	@Home	Edit or Delete or Edit Resources

O uso de roteamentos similares a esse nos permite criar URLs que poderão ser facilmente identificadas e deduzidas pelo usuário. Isso é especialmente útil na criação de *permalinks*, ou seja, links que identificam unicamente um recurso e não mudam.

Esse tipo de customização é especialmente útil para classes de URL onde uma identificação mais textual é necessária como, por exemplo, URLs que definem páginas em um gerenciador de conteúdo, ou URLs que

definem *posts* em um *blog*.

Existe ainda uma outra classe de roteamentos que são chamados de *named routes* no Rails. Esse tipo de roteamento é chamado assim porque o resultado é uma método com um nome específico que pode ser invocado de *controllers* e *views*. Um exemplo seria modificar o nosso arquivo de roteamentos para:

```
ActionController::Routing::Routes.draw do |map|
  # The priority is based upon order of creation: first created -> highest priority.

  # Sample of regular route:
  # map.connect "products/:id", :controller => "catalog", :action => "view"
  # Keep in mind you can assign values other than :controller and :action

  # Sample of named route:
  # map.purchase "products/:id/purchase", :controller => "catalog", :action => "purchase"
  # This route can be invoked with purchase_url(:id => product.id)

  # You can have the root of your site routed by hooking up ""
  # -- just remember to delete public/index.html.
  map.home "", :controller => "home"

  # Allow downloading Web Service WSDL as a file with an extension
  # instead of a file named "wsdl"
  map.connect ":controller/service.wsdl", :action => "wsdl"

  map.connect ":year/:month/:day", :controller => "actions", :action => "by_date",
    :requirements => {
      :year => /\d{4}/,
      :day => /\d{1,2}/,
      :month => /\d{1,2}/
    }

  # Install the default route as the lowest priority.
  map.connect ":controller/:action/:id"
end
```

Note que, ao invés de usarmos o método `connect`, estamos dando um nome específico para o nosso roteamento. Como dito anteriormente, isso permite que o mesmo seja descrito em *controllers* e *views* por esse nome. Digamos, por exemplo, que queremos agora adicionar uma *link* para a *home* da aplicação em nossa cabeçalho. Poderíamos fazer algo assim em nosso *layout* primário:

```
<html>

<head>
  <title>GTD</title>
  <%= stylesheet_link_tag "default" %>
  <%= stylesheet_link_tag "scaffold" %>
  <%= javascript_include_tag :defaults %>
</head>

<body>

  <h1 id="header"><%= link_to "GTD", home_url %></h1>

  <% if session_user %>
  <p id="login-information">
    You are logged in as <strong><%= session_user.name %></strong>.
    <%= link_to "Log out", :controller => "login", :action => "logout" %>.
  </p>
```

```

<% end %>

<%= image_tag("indicator.gif", :id => "indicator", :style => "float: right; margin: 5px; display: none") %>



- <%= link_to_unless_current "&raquo; Contexts", :controller => "contexts", :action => "list" %>
- <%= link_to_unless_current "&raquo; Projects", :controller => "projects", :action => "list" %>
- <%= link_to_unless_current "&raquo; Actions", :controller => "actions", :action => "list" %>
- <%= link_to_unless_current "&raquo; Resources", :controller => "resources", :action => "list" %>



<%= yield %>



</body>

</html>

```

Quando definimos uma roteamento nomeado, o Rails automaticamente cria um método descrevendo o mesmo com o sufixo `_url`, como você pode ver acima. Essa é mais uma facilidade oferecida pelo *framework* para facilitar o trabalho do desenvolvedor.

Para completar a nossa modificação, basta inserir o seguinte fragmento em nosso arquivo `default.css`:

```

h1#header a
{
  text-decoration: none;
  color: inherit;
}

```

Pode não parecer muito a princípio, mas, combinado isso com parâmetros, temos algo bastante sofisticado. Veja o exemplo abaixo, mostrando um arquivo `routes.rb` hipotético:

```

ActionController::Routing::Routes.draw do |map|
  map.with_options :controller => "blog" do |blog|
    blog.show "", :action => "list"
    blog.delete "delete/:id", :action => "delete",
    blog.edit "edit/:id", :action => "edit"
  end
  map.connect ':controller/:action/:view'
end

```

Com o arquivo abaixo poderíamos usar métodos como:

```

<%= link_to @blog.title, show_url %>
<%= link_to @blog.title, delete_url(:id => @blog.id) %>

```

Como podemos ver, isso realmente pode facilitar a nossa vida. O único inconveniente é a poluição de nosso *namespace* com dezenas de métodos desse tipo. O mais interessante provavelmente é fazer isso para métodos bastante usados.

CACHING

Quando mencionamos otimizações anteriormente, indicamos o fato de que existem soluções próprias no Rails para alguns problemas comuns. Uma vez que sua aplicação esteja terminada e você tenha identificado pontos que podem se beneficiar de ajustes de performance, algumas soluções dadas pelo Rails podem ser de bastante ajuda.

Uma dessas soluções é o mecanismo de *caching* existente no Rails, que permite que páginas inteiras ou partes de páginas sejam renderizadas e guardadas como prontas para exibição futura sem necessidade de executar o código responsável pela mesma novamente até que ela seja invalidada por alguma mudança qualquer.

O mecanismo de *caching* possui uma série de funcionalidades que permitem um bom grau de flexibilidade como, por exemplo, criar várias versões de um *cache* dependendo dos parâmetros recebidos por uma página.

Apesar disso, um dos grandes problemas de se usar *caching* em Rails é que é relativamente difícil acertar as opções corretas sem testes cuidadosos. Existem três tipos principais de *caches* no Rails e cada um tem suas vantagens e desvantagens. A presente seção fornece uma visão geral de como você pode usar esses métodos sem muitas complicações, apontando também os problemas de cada um. Hoje, com o crescimento do uso de *cache* em aplicações, *plugins* estão surgindo para corrigir alguns dos problemas existentes do Rails, mas há ainda alguns problemas que exigem uma aproximação mais manual.

Para testar essas opções, vamos começar fazendo o *cache* de uma fragmento de página. Vamos usar a ação *list* do *controller* *actions* como exemplo inicial. Antes de usarmos o mecanismo de *caching*, porém, precisamos habilitá-lo no modo de desenvolvimento, já que ele somente é habilitado por padrão em produção. Você pode modificar o arquivo *config/environments/development.rb* para isso:

```
# Settings specified here will take precedence over those in config/environment.rb

# In the development environment your application's code is reloaded on
# every request. This slows down response time but is perfect for development
# since you don't have to restart the webserver when you make code changes.
config.cache_classes = false

# Log error messages when you accidentally call methods on nil.
config.whiny_nils = true

# Enable the breakpoint server that script/breakpointer connects to
config.breakpoint_server = true

# Show full error reports and disable caching
config.action_controller.consider_all_requests_local = true
```

```

config.action_controller.perform_caching = true
config.action_view.cache_template_extensions = false
config.action_view.debug_rjs = true

# Don't care if the mailer can't send
config.action_mailer.raise_delivery_errors = false

```

Reinic peace o servidor para que as modificações tenham efeito.

Agora podemos modificar a nossa *view* para ter algum *cache*:

```

<h1>Actions</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New Action", :action => "edit" %></p>

<% cache do %>
<table border="1" cellpadding="5" cellspacing="1">
  <tr>
    <th>Description</th>
    <th>Completed</th>
    <th>When</th>
    <th>Project</th>
    <th>Context</th>
    <th>Actions</th>
  </tr>
  <% @actions.each do |action| %>
  <tr>
    <td><%= action.description %></td>
    <td><%= yes_or_no?(action.done?) %></td>
    <td><%= (action.done?) ? action.completed_at.strftime("%m/%d/%y") : "-" %></td>
    <td><%= action.project.name %></td>
    <td><%= action.context.name %></td>
    <td nowrap>
      <%= link_to "Edit", :action => "edit", :id => action %> or<br>
      <%= link_to "Delete", { :action => "delete", :id => action }, :confirm => "Are you sure?" %>
    or<br>
      <%= link_to "Edit Resources", :action => "resources", :id => action %>
    </td>
  </tr>
  <% end %>
</table>
<% end %>

<p><%= pagination_links(@action_pages) %></p>

```

No caso acima, estamos fazendo o *cache* de um fragmento da página. Se observamos agora o diretório `tmp/cache` sob nossa aplicação, veremos que arquivos foram criados para conter a renderização pronta do fragmento.

Se adicionarmos um novo objeto ao banco agora, veremos um fato curioso: ele não aparece em nossa listagem. Outra coisa que notaremos é que, se logarmos como um outro usuário, veremos as ações do primeiro. Isso acontece porque o *cache* não foi renovado no momento em que o banco sofreu modificações.

Podemos corrigir o primeiro problema expirando o *cache* ao inserirmos, modificarmos ou excluirmos um

objeto. Um arquivo de *cache* está automaticamente associado com o *controller* e ação onde foi criado. Por isso, podemos mudar o nosso *controller* da seguinte maneira:

```
class ActionsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @action_pages, @actions = paginate :actions, :include => [:project, :context], :per_page => 5,
    :order => "actions.description"
  end

  def by_date
    @action_pages, @actions = paginate :actions, :conditions => ["actions.created_at = ?",
    Date.new(params[:year].to_i, params[:month].to_i, params[:day].to_i)], :include => [:project,
    :context], :per_page => 5, :order => "actions.description"
    render :action => "list"
  end

  def edit
    @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    @projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    if request.get?
      if params[:id]
        @item = Action.find(params[:id])
      else
        @item = Action.new
      end
    elsif request.post?
      @item = params[:id] ?
        Action.update(params[:id], params[:item]) :
        Action.create(params[:item])
      if @item.valid?
        expire_fragment(:action => "list")
        flash[:notice] = "The action was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def resources
    @action = Action.find(params[:id])
    @resources = []
    if request.post? && !params[:keywords].blank?
      @resources = Resource.find(:all, :conditions => ['name like ? or filename like ?',
      "%#{params[:keywords]}%", "%#{params[:keywords]}%"])
    end
  end

  def add_resources
    @action = Action.find(params[:id])
    existing_resource_ids = @action.resources.collect { |resource| resource.id }
    new_resource_ids = params[:resources].reject { |id| existing_resource_ids.include?(id.to_i) }
    @action.resources << Resource.find(new_resource_ids)
    redirect_to :action => "resources", :id => @action.id
  end

  def delete_resource
    @action = Action.find(params[:id])
    @action.resources.delete(Resource.find(params[:resource_id]))
    redirect_to :action => "resources", :id => @action.id
  end

  def delete
    Action.find(params[:id]).destroy
  end

```

```

flash[:notice] = "The action was successfully deleted"
redirect_to :action => "list"
expire_fragment(:action => "list")
end
end

```

Essas modificações forçam o *cache* a ser renovado quando há mudanças nos dados existentes no banco e agora é possível ver que a página é atualiza, embora persista o problema de dados incorretos aparecendo para outro usuário. Para corrigir o problema, precisamos de duas mudanças.

Primeiro em nossa nossa *view*, onde teríamos algo assim:

```

<h1>Actions</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New Action", :action => "edit" %></p>

<% cache(:action => "list", :id => session[:user]) do %>
<table border="1" cellpadding="5" cellspacing="1">
<tr>
<th>Description</th>
<th>Completed</th>
<th>When</th>
<th>Project</th>
<th>Context</th>
<th>Actions</th>
</tr>
<% @actions.each do |action| %>
<tr>
<td><%= action.description %></td>
<td><%= yes_or_no?(action.done?) %></td>
<td><%= (action.done?) ? action.completed_at.strftime("%m/%d/%y") : "-" %></td>
<td><%= action.project.name %></td>
<td><%= action.context.name %></td>
<td nowrap>
<%= link_to "Edit", :action => "edit", :id => action %> or<br>
<%= link_to "Delete", { :action => "delete", :id => action }, :confirm => "Are you sure?" %>
or<br>
<%= link_to "Edit Resources", :action => "resources", :id => action %>
</td>
</tr>
<% end %>
</table>
<% end %>

<p><%= pagination_links(@action_pages) %></p>

```

Explicitando o fragmento que estamos gravando em *cache* e informando o *id* do usuário logado, somos capaz de gerar fragmentos customizados para cada pessoa acessando o sistema, aumentando a performance de nossas aplicação sem mais do que algumas pequenas modificações. Uma listagem do diretório onde o Rails grava o *cache* releva o seguinte, depois de algumas invocações:

```

ronaldo@minerva:~/tmp/gtd/tmp/cache/localhost.3000/actions/list$ ls -l
total 8

```

```
-rw-r--r-- 1 ronaldo ronaldo 1957 2006-10-01 02:02 1.cache
-rw-r--r-- 1 ronaldo ronaldo 1188 2006-10-01 02:01 2.cache
```

Como podemos ver, há um arquivo para cada usuário no *cache*.

Agora precisamos mudar a forma como expiramos fragmentos, alterando nosso *controller*.

```
class ActionsController < ApplicationController

  def index
    list
    render :action => "list"
  end

  def list
    @action_pages, @actions = paginate :actions, :include => [:project, :context], :per_page => 5,
    :order => "actions.description"
    end

  def by_date
    @action_pages, @actions = paginate :actions, :conditions => ["actions.created_at = ?",
    Date.new(params[:year].to_i, params[:month].to_i, params[:day].to_i)], :include => [:project,
    :context], :per_page => 5, :order => "actions.description"
    render :action => "list"
  end

  def edit
    @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    @projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    if request.get?
      if params[:id]
        @item = Action.find(params[:id])
      else
        @item = Action.new
      end
    elsif request.post?
      @item = params[:id] ?
        Action.update(params[:id], params[:item]) :
        Action.create(params[:item])
      if @item.valid?
        expire_fragment(:action => "list", :id => session[:user])
        flash[:notice] = "The action was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def resources
    @action = Action.find(params[:id])
    @resources = []
    if request.post? && !params[:keywords].blank?
      @resources = Resource.find(:all, :conditions => ['name like ? or filename like ?',
      "%#{params[:keywords]}%", "%#{params[:keywords]}%"])
    end
  end

  def add_resources
    @action = Action.find(params[:id])
    existing_resource_ids = @action.resources.collect { |resource| resource.id }
    new_resource_ids = params[:resources].reject { |id| existing_resource_ids.include?(id.to_i) }
    @action.resources << Resource.find(new_resource_ids)
    redirect_to :action => "resources", :id => @action.id
  end

  def delete_resource
    @action = Action.find(params[:id])
```

```

@action.resources.delete(Resource.find(params[:resource_id]))
redirect_to :action => "resources", :id => @action.id
end

def delete
  Action.find(params[:id]).destroy
  flash[:notice] = "The action was successfully deleted"
  redirect_to :action => "list"
  expire_fragment(:action => "list", :id => session[:user])
end
end

```

Note que não estamos fazendo *cache* da paginação. O problema é similar ao que acontecia quando acessávamos os dados com um usuário diferente. Se você adicionar mais do que um certo número de ações para que a paginação apareça verá que a primeira listagem em *cache* será retornada para todas as chamadas. A solução é similar também.

Primeiro, modificamos a nossa *view*:

```

<h1>Actions</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New Action", :action => "edit" %></p>

<% cache(:action => "list", :id => session[:user], :page => page) do %>

  <table border="1" cellpadding="5" cellspacing="1">
    <tr>
      <th>Description</th>
      <th>Completed</th>
      <th>When</th>
      <th>Project</th>
      <th>Context</th>
      <th>Actions</th>
    </tr>
    <% @actions.each do |action| %>
    <tr>
      <td><%= action.description %></td>
      <td><%= yes_or_no?(action.done?) %></td>
      <td><%= (action.done?) ? action.completed_at.strftime("%m/%d/%Y") : "-" %></td>
      <td><%= action.project.name %></td>
      <td><%= action.context.name %></td>
      <td nowrap>
        <%= link_to "Edit", :action => "edit", :id => action %> or<br>
        <%= link_to "Delete", { :action => "delete", :id => action }, :confirm => "Are you sure?" %>
      or<br>
        <%= link_to "Edit Resources", :action => "resources", :id => action %>
      </td>
    </tr>
    <% end %>
  </table>

  <p><%= pagination_links(@action_pages) %></p>

<% end %>

```

Isso resolve a craga de dados diferentes para outras páginas da listagem. Se você observar o diretório de *cache* verá o seguinte:

```
ronaldo@minerva:~/tmp/gtd/tmp/cache/localhost.3000/actions/list$ ls -l
total 12
-rw-r--r-- 1 ronaldo ronaldo 2076 2006-10-01 10:07 1.page=1.cache
-rw-r--r-- 1 ronaldo ronaldo 602 2006-10-01 10:07 1.page=2.cache
-rw-r--r-- 1 ronaldo ronaldo 1251 2006-10-01 10:07 2.page=1.cache
```

Agora só precisamos alterar o nosso *controller* para expirar todas as páginas relacionadas ao usuário se houver uma mudança e introduzir o método *page* que usamos:

```
class ActionsController < ApplicationController

  helper_method :page

  def index
    list
    render :action => "list"
  end

  def list
    @action_pages, @actions = paginate :actions, :include => [:project, :context], :per_page => 5,
    :order => "actions.description"
    end

    def by_date
      @action_pages, @actions = paginate :actions, :conditions => ["actions.created_at = ?",
      Date.new(params[:year].to_i, params[:month].to_i, params[:day].to_i)], :include => [:project,
      :context], :per_page => 5, :order => "actions.description"
      render :action => "list"
    end

    def edit
      @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
      @projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
      if request.get?
        if params[:id]
          @item = Action.find(params[:id])
        else
          @item = Action.new
        end
      elsif request.post?
        @item = params[:id] ?
          Action.update(params[:id], params[:item]) :
          Action.create(params[:item])
        if @item.valid?
          expire_fragment('/actions/list/#{$session[:user]}(.*)')
          flash[:notice] = "The action was successfully saved"
          redirect_to :action => "list"
        end
      end
    end

    def resources
      @action = Action.find(params[:id])
      @resources = []
      if request.post? && !params[:keywords].blank?
        @resources = Resource.find(:all, :conditions => ['name like ? or filename like ?',
        "%#{params[:keywords]}%", "%#{params[:keywords]}%"])
      end
    end

    def add_resources
      @action = Action.find(params[:id])
      existing_resource_ids = @action.resources.collect { |resource| resource.id }
      new_resource_ids = params[:resources].reject { |id| existing_resource_ids.include?(id.to_i) }
      @action.resources << Resource.find(new_resource_ids)
```

```

    redirect_to :action => "resources", :id => @action.id
  end

  def delete_resource
    @action = Action.find(params[:id])
    @action.resources.delete(Resource.find(params[:resource_id]))
    redirect_to :action => "resources", :id => @action.id
  end

  def delete
    Action.find(params[:id]).destroy
    flash[:notice] = "The action was successfully deleted"
    redirect_to :action => "list"
    expire_fragment(/actions\/list\/#{session[:user]}(.*)/)
  end

  protected

  def page
    params[:page] || 1
  end

end

```

Usamos aqui uma expressão regular, customizada por usuário, para remover qualquer arquivo de *cache* que esteja relacionado ao mesmo. Não podemos usar diretamente o parâmetro `page` porque o método `expire_fragment`, quando usado com parâmetros livres, opera sobre fragmentos específicos.

Obviamente, se você está customizando o seu *cache* a esse ponto, tenha certeza de que é isso o que você precisa realmente porque, caso contrário, ao invés de diminuir a carga no servidor, você estará aumentando a mesma, considerando que o Rails passa a ter que controlar dezenas ou centenas de páginas em *cache*. De uma forma geral, **não** é interessante fazer *cache* baseado em parâmetros que mudam freqüentemente—principalmente no caso de buscas—mas a análise deve ser feita caso a caso.

A desvantagem de usar *cache* por fragmentos é que ele se aplica somente a *views*. Se você observar o *log* do servidor durante a renderização da ação, verá que a chamada ao banco ainda é feita mesmo que a página já esteja em *cache*.

```

Processing ActionsController#list (for 127.0.0.1 at 2006-10-01 11:00:00) [GET]
Session ID: 446574c6059de1924c9bc2b1cb73374e
Parameters: {"action"=>"list", "controller"=>"actions"}
Action Count (0.006723)   SELECT COUNT(DISTINCT actions.id) FROM actions LEFT OUTER JOIN projects
ON projects.id = actions.project_id LEFT OUTER JOIN contexts ON contexts.id = actions.context_id
WHERE (actions.user_id = 1)
Action Columns (0.002482)   SHOW FIELDS FROM actions
Project Columns (0.002172)   SHOW FIELDS FROM projects
Context Columns (0.001632)   SHOW FIELDS FROM contexts
Action Load Including Associations (0.002830)   SELECT actions.`id` AS t0_r0, actions.`description` AS t0_r1, actions.`done` AS t0_r2, actions.`created_at` AS t0_r3, actions.`completed_at` AS t0_r4, actions.`context_id` AS t0_r5, actions.`project_id` AS t0_r6, actions.`user_id` AS t0_r7, actions.`position` AS t0_r8, projects.`id` AS t1_r0, projects.`name` AS t1_r1, projects.`description` AS t1_r2, projects.`active` AS t1_r3, projects.`user_id` AS t1_r4, contexts.`id` AS t2_r0, contexts.`name` AS t2_r1, contexts.`user_id` AS t2_r2 FROM actions LEFT OUTER JOIN projects ON projects.id = actions.project_id LEFT OUTER JOIN contexts ON contexts.id = actions.context_id WHERE (actions.user_id = 1) ORDER BY actions.description LIMIT 0, 5
Rendering within layouts/application
Rendering actions/list
Fragment read: localhost:3000/actions/list/1?page=1 (0.00021)

```

```
User Load (0.001003)    SELECT * FROM users WHERE (id = 1) LIMIT 1
User Columns (0.003208)    SHOW FIELDS FROM users
Completed in 0.04208 (23 reqs/sec) | Rendering: 0.00922 (21%) | DB: 0.02005 (47%) | 200 OK
[http://localhost/actions/list]
```

Como é fácil perceber, mesmo com a leitura de um fragmento pelo mecanismo de *caching*, o banco de dados continua a ser acessado. Não existe uma maneira padrão de resolver isso, especialmente quando estamos lidando com fragmentos que dependem de várias parâmetros—como é o nosso caso no momento.

Uma solução possível ser customizar a invocação ao banco de modo que ela só ocorra quando necessário, ou seja, quando o fragmento não for lido. Um exemplo disso seria fazer as modificações vistas em seguida.

Primeiro, modificamos a *view* para não usar variáveis de instância, mas usar métodos, permitindo um maior controle sobre a saída:

```
<h1>Actions</h1>

<% if flash[:notice] %>
<p style="color: green; font-style: italic"><%= flash[:notice] %></p>
<% end %>

<p><%= link_to "New Action", :action => "edit" %></p>

<% cache(:action => "list", :id => session[:user], :page => page) do %>

  <table border="1" cellpadding="5" cellspacing="1">
    <tr>
      <th>Description</th>
      <th>Completed</th>
      <th>When</th>
      <th>Project</th>
      <th>Context</th>
      <th>Actions</th>
    </tr>
    <% actions.each do |action| %>
    <tr>
      <td><%= action.description %></td>
      <td><%= yes_or_no?(action.done?) %></td>
      <td><%= (action.done?) ? action.completed_at.strftime("%m/%d/%y") : "-" %></td>
      <td><%= action.project.name %></td>
      <td><%= action.context.name %></td>
      <td nowrap>
        <%= link_to "Edit", :action => "edit", :id => action %> or<br>
        <%= link_to "Delete", { :action => "delete", :id => action }, :confirm => "Are you sure?" %>
      or<br>
        <%= link_to "Edit Resources", :action => "resources", :id => action %>
      </td>
    </tr>
    <% end %>
  </table>

  <p><%= pagination_links(action_pages) %></p>
<% end %>
```

Depois disso, modificamos o *controller* para definir esses métodos:

```
class ActionsController < ApplicationController
```

```

helper_method :page
helper_method :actions, :action_pages

def index
  list
  render :action => "list"
end

def list
end

def by_date
  @action_pages, @actions = paginate :actions, :conditions => ["actions.created_at = ?",
Date.new(params[:year].to_i, params[:month].to_i, params[:day].to_i)], :include => [:project,
:context], :per_page => 5, :order => "actions.description"
  render :action => "list"
end

def edit
  @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
  @projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
  if request.get?
    if params[:id]
      @item = Action.find(params[:id])
    else
      @item = Action.new
    end
  elsif request.post?
    @item = params[:id] ?
      Action.update(params[:id], params[:item]) :
      Action.create(params[:item])
    if @item.valid?
      expire_fragment(/actions\/list\//#{session[:user]}(.*)/)
      flash[:notice] = "The action was successfully saved"
      redirect_to :action => "list"
    end
  end
end

def resources
  @action = Action.find(params[:id])
  @resources = []
  if request.post? && !params[:keywords].blank?
    @resources = Resource.find(:all, :conditions => ['name like ? or filename like ?',
"%#{params[:keywords]}%", "%#{params[:keywords]}%"])
  end
end

def add_resources
  @action = Action.find(params[:id])
  existing_resource_ids = @action.resources.collect { |resource| resource.id }
  new_resource_ids = params[:resources].reject { |id| existing_resource_ids.include?(id.to_i) }
  @action.resources << Resource.find(new_resource_ids)
  redirect_to :action => "resources", :id => @action.id
end

def delete_resource
  @action = Action.find(params[:id])
  @action.resources.delete(Resource.find(params[:resource_id]))
  redirect_to :action => "resources", :id => @action.id
end

def delete
  Action.find(params[:id]).destroy
  flash[:notice] = "The action was successfully deleted"
  redirect_to :action => "list"
  expire_fragment(/actions\/list\//#{session[:user]}(.*)/)
end

protected

```

```

def actions
  load_listing
  @actions
end

def action_pages
  load_listing
  @action_pages
end

def load_listing
  unless @actions
    @action_pages, @actions = paginate :actions, :include => [:project, :context], :per_page =>
5, :order => "actions.description"
  end
end

def page
  params[:page] || 1
end

```

No código acima, removemos as invocações de diretamente da ação `list` do *controller*. Usamos então um método, `load_listing`, que carrega os nossos dados somente se isso não tivesse acontecido antes. Esse método é usado internamente pelos outros dois métodos que são definidos para serem os *helpers* a serem usados na *view*. O resultado podemos ver no *log* na próxima invocação da página:

```

Processing ActionsController#list (for 127.0.0.1 at 2006-10-01 18:07:24) [GET]
Session ID: ead83373402f4ec08ed116190d655f3f
Parameters: {"action"=>"list", "controller"=>"actions", "page"=>"1"}
Rendering within layouts/application
Rendering actions/list
Fragment read: localhost:3000/actions/list/1?page=1 (0.00031)
User Load (0.001832)  SELECT * FROM users WHERE (id = 1) LIMIT 1
User Columns (0.004119)  SHOW FIELDS FROM users
Completed in 0.03208 (31 reqs/sec) | Rendering: 0.01904 (59%) | DB: 0.00595 (18%) | 200 OK
[http://localhost/actions/list?page=1]

```

Podemos ver agora que as chamadas ao banco para carregar ações não são mais invocadas exceto quando o *cache* é expirado.

A solução que descrevemos aqui obviamente não deve ser usada indiscriminadamente. Ela é meramente uma descrição de uma estratégia possível.

Uma outra maneira de fazer o *cache* de páginas, criada para resolver o problema de acesso ao banco acima, é fazer o *cache* da página inteira. Existem dois métodos para isso no Rails, `caches_page` e `caches_action`. Os dois funcionam de maneira similar, fazendo o *cache* de toda a página. A diferença é que no caso do primeiro, `caches_page`, a requisição nem passa pelo Rails uma vez que a página tenha ido para o *cache*. O arquivo gerado de cache é servido diretamente do servidor. Em outras palavras, filtros não são executados, *callbacks* não rodam e assim por diante até que você expire a página no *cache*.

Obviamente o comportamento de `caches_page` não é útil para ações que dependem de customizações em

tempo de execução baseadas em parâmetros e autenticações. Mas pode ser extremamente útil para páginas públicas onde o conteúdo varia pouco, especialmente quanto as mesmas são resultado de acessos intensivos de banco.

Vamos experimentar agora o método `caches_action` com uma ação que não seja individualizada por usuário, como é o próprio cadastro de usuários. O método é aplicado no próprio *controller*, com a descrição da ação, como podemos ver abaixo:

```
class UsersController < ApplicationController
  caches_action :list
  before_filter :authenticate_administration

  def index
    list
    render :action => "list"
  end

  def list
    @user_pages, @users = paginate :users, :per_page => 5, :order => "name"
  end

  def edit
    if params[:id]
      @user = User.find(params[:id])
    else
      @user = User.new
    end
    if request.post?
      @user.attributes = params[:user]
      send_email = @user.new_record?
      if @user.save
        expire_action(:action => "list")
        if send_email
          RegistrationNotifier.deliver_registration_notification(@user, session_user)
        end
        flash[:notice] = "The user was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def delete
    User.find(params[:id]).destroy
    flash[:notice] = "The user was successfully deleted"
    redirect_to :action => "list"
    expire_action(:action => "list")
  end
end
```

Vemos também que, como em nosso exemplo anterior, precisamos de uma chamada para invalidar o *cache*.

O código acima funciona, mas tem dois problema sérios. Primeiro, quando você inclui, altera ou exclui um registro, a mensagem gerada na variável `flash` se torna parte do arquivo em *cache*. Isso acontece porque estamos fazendo um *cache* do conteúdo completo gerado pela ação e não somente de um fragmento. Segundo, o problema com paginação também acontece.

Nenhum desses dois problemas pode ser resolvido com modificações à chamada do método `caches_action` porque o mesmo não suporta parâmetros que permitem customizar as chamadas. Esse é um exemplo de como você tem que ser cuidadoso no uso de *caching* para não provocar estranhos efeitos colaterais.

Mesmo assim, há soluções para ambos os problemas.

A questão das variáveis *flash*, por exemplo, poderia ser resolvida como uma modificação dupla, que somente descreveremos aqui: primeiro, movemos o código que exibe a mensagem *flash* para o nosso *layout*, o que poderia ser feito sem prejuízo para a aplicação; segundo, fazemos uma invocação do *cache* através de um *plugin* chamado *content_cache* que pode ser usado para salvar somente o conteúdo da *view* e não da página por completo. Esse *plugin* e uma explicação detalhada de como ele funcionam, além de instruções de uso, podem ser encontradas no endereço: <http://blog.codahale.com/2006/04/10/content-only-caching-for-rails/>.

Uma outra solução para o mesmo problema seria escrever o seu próprio filtro. O método `caches_actions` é nada mais do que uma invocação a um filtro *around* e você poderia escrever um similar com bem pouco esforço.

Para o problema de paginação, a solução seria criar um roteamento especializada que nos permitiria gerar a URL customizada para cada página da listagem de modo que elas aparecessem para o *cache* com ações diferentes. Bastaria acrescentar algo assim ao arquivo `routes.rb`:

```
ActionController::Routing::Routes.draw do |map|
  # The priority is based upon order of creation: first created -> highest priority.

  # Sample of regular route:
  # map.connect "products/:id", :controller => "catalog", :action => "view"
  # Keep in mind you can assign values other than :controller and :action

  # Sample of named route:
  # map.purchase "products/:id/purchase", :controller => "catalog", :action => "purchase"
  # This route can be invoked with purchase_url(:id => product.id)

  # You can have the root of your site routed by hooking up ""
  # -- just remember to delete public/index.html.
  map.home "", :controller => "home"

  # Allow downloading Web Service WSDL as a file with an extension
  # instead of a file named "wsdl"
  map.connect ":controller/service.wsdl", :action => "wsdl"

  map.connect ":year/:month/:day", :controller => "actions", :action => "by_date",
    :requirements => {
      :year => /\d{4}/,
      :day => /\d{1,2}/,
      :month => /\d{1,2}/
    }

  map.connect "users/list/:page", :controller => "users", :action => "list"

  # Install the default route as the lowest priority.
  map.connect ":controller/:action/:id"
end
```

O resultado seriam URLs no seguinte formato:

```
http://localhost:3000/users/list/1  
http://localhost:3000/users/list/2  
...
```

Pela implementação atual do método `caches_action`, cada URL acima geraria seu próprio `cache`, resolvendo o problema de paginação, embora não o problema das variáveis `flash`.

Um outro inconveniente do usar `cache` é que precisamos controlar a expiração das páginas, um trabalho que pode ficar tedioso em pouco tempo. Para resolver o problema, o Rails possui uma espécie de classes chamadas `Sweeper` que podem ser usada para automaticamente monitorar objetos e expirar ações e fragmentos de acordo com a necessidade.

Vamos supor que estamos usando a estratégia ousada para ações em nossos outros `controllers`. Poderíamos criar uma classe `Sweeper` genérica que nos permitisse expirar o `cache` automaticamente, sem código extra no `controller`. Como queremos uma classe genérica, vamos criar um arquivo novo no diretório `extras`, chamado `list_sweeper.rb`:

```
class ListSweeper < ActionController::Caching::Sweeper  
  observe Project, Context, Action, Resource  
  
  def after_save(record)  
    expire_record(record)  
  end  
  
  def after_destroy(record)  
    expire_record(record)  
  end  
  
  def expire_record(record)  
    expire_fragment(/#{record.class.table_name}\list\#{session[:user]}(.*)/) /  
  end  
end
```

O que essa classe faz é observar mudanças nas classes de dados especificadas acima, respondendo a duas delas.

O método `observe`, que usamos acima, é definido na classe `Observer`, da qual a classe `Sweeper` é derivada e serve para acompanhar de forma automática acessos ao banco de dados feitos por uma classe através dos `callbacks` que vimos anteriormente.

Em nossa classe acima, estamos monitorando o momento em que um objeto é salvo ou excluído, removendo o `cache` do mesmo. Como o objeto observado é passado ao `callback`, temos todos seus dados disponíveis. É por meio disso que estamos usando `record.class.table_name` para descobrir o diretório de `cache` necessário, considerando que em nossa aplicação o nome da tabela é mapeado perfeitamente para nossos `controllers`.

Precisamos agora incluir a nova biblioteca em nosso arquivo environment.rb:

```
# Be sure to restart your web server when you modify this file.

# Uncomment below to force Rails into production mode when
# you don't control web/app server and can't set it the proper way
# ENV['RAILS_ENV'] ||= 'production'

# Specifies gem version of Rails to use when vendor/rails is not present
RAILS_GEM_VERSION = '1.1.6'

# Bootstrap the Rails environment, frameworks, and default configuration
require File.join(File.dirname(__FILE__), 'boot')

Rails::Initializer.run do |config|
  # Settings in config/environments/* take precedence those specified here

  # Skip frameworks you're not going to use (only works if using vendor/rails)
  # config.frameworks -= [ :action_web_service, :action_mailer ]

  # Add additional load paths for your own custom dirs
  config.load_paths += %W( #{RAILS_ROOT}/extras )

  # Force all environments to use the same logger level
  # (by default production uses :info, the others :debug)
  # config.log_level = :debug

  # Use the database for sessions instead of the file system
  # (create the session table with 'rake db:sessions:create')
  # config.action_controller.session_store = :active_record_store

  # Use SQL instead of Active Record's schema dumper when creating the test database.
  # This is necessary if your schema can't be completely dumped by the schema dumper,
  # like if you have constraints or database-specific column types
  # config.active_record.schema_format = :sql

  # Activate observers that should always be running
  # config.active_record.observers = :cacher, :garbage_collector

  # Make Active Record use UTC-base instead of local time
  # config.active_record.default_timezone = :utc

  # See Rails::Configuration for more options
end

# Add new inflection rules using the following format
# (all these examples are active by default):
# Inflector.inflections do |inflect|
#   inflect.plural /^(ox)$/, '\1en'
#   inflect.singular /^(ox)en/, '\1'
#   inflect.irregular 'person', 'people'
#   inflect.uncountable %w( fish sheep )
# end

# Include your application configuration below
require "user_filter.rb"
require "auditing.rb"
require "acts_as_drop_down.rb"
require "list_sweeper.rb"

ActionMailer::Base.server_settings =
{
  :address => "mail.yourdomain.com",
  :domain => "yourdomain.com",
  :user_name => "user@yourdomain.com",
  :password => "*****",
  :authentication => :login
```

```
}
```

Reinic peace o servidor para que as modificações tenham efeito.

Agora modificamos o nosso *controller* primário, em `application.rb`, para usar a classe acima:

```
class ApplicationController < ActionController::Base

  before_filter :authenticate
  before_filter :set_current_user_id

  skip_before_filter :authenticate_administration, :only => [:access_denied]

  around_filter UserFilter.new(Context, :needed_scopes)
  around_filter UserFilter.new(Project, :needed_scopes)
  around_filter UserFilter.new(Action, :needed_scopes)
  around_filter UserFilter.new(Resource, :needed_scopes)

  helper_method :session_user

  cache_sweeper :list_sweeper, :only => [ :edit, :delete ]

  def access_denied
    render :template => "shared/access_denied"
  end

  protected

  def set_current_user_id
    User.current_user_id = session[:user]
  end

  def session_user
    @session_user ||= User.find(:first, :conditions => ['id = ?', session[:user]])
  end

  def authenticate
    unless session[:user]
      session[:return_to] = request.request_uri
      redirect_to :controller => "login", :action => "login"
      return false
    end
    return true
  end

  def authenticate_administration
    unless session_user && session_user.admin?
      redirect_to :action => "access_denied"
      return false
    end
    return true
  end

  def needed_scopes(target_class)
  {
    :find => { :conditions => ["#{target_class.table_name}.user_id = ?", session[:user]] },
    :create => { :user_id => session[:user] }
  }
end
```

A linha acima indica que o *controller* deve registrar um *sweeper* para as ações `edit` e `delete`. Quando essas ações forem executadas, o registro será modificado, disparando a classe que registramos e limpando o

cache como queremos.

Você pode testar o funcionamento do código, removendo o código de expiração do *controller* que estamos usando, retornando-o ao original:

```
class ActionsController < ApplicationController

  helper_method :page
  helper_method :actions, :action_pages

  def index
    list
    render :action => "list"
  end

  def list
  end

  def by_date
    @action_pages, @actions = paginate :actions, :conditions => ["actions.created_at = ?",
Date.new(params[:year].to_i, params[:month].to_i, params[:day].to_i)], :include => [:project,
:context], :per_page => 5, :order => "actions.description"
    render :action => "list"
  end

  def edit
    @contexts = Context.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    @projects = Project.find(:all, :order => "name").collect { |i| [i.name, i.id] }
    if request.get?
      if params[:id]
        @item = Action.find(params[:id])
      else
        @item = Action.new
      end
    elsif request.post?
      @item = params[:id] ?
        Action.update(params[:id], params[:item]) :
        Action.create(params[:item])
      if @item.valid?
        flash[:notice] = "The action was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def resources
    @action = Action.find(params[:id])
    @resources = []
    if request.post? && !params[:keywords].blank?
      @resources = Resource.find(:all, :conditions => ['name like ? or filename like ?',
"%#{params[:keywords]}%", "%#{params[:keywords]}%"])
    end
  end

  def add_resources
    @action = Action.find(params[:id])
    existing_resource_ids = @action.resources.collect { |resource| resource.id }
    new_resource_ids = params[:resources].reject { |id| existing_resource_ids.include?(id.to_i) }
    @action.resources << Resource.find(new_resource_ids)
    redirect_to :action => "resources", :id => @action.id
  end

  def delete_resource
    @action = Action.find(params[:id])
    @action.resources.delete(Resource.find(params[:resource_id]))
    redirect_to :action => "resources", :id => @action.id
  end
```

```

def delete
  Action.find(params[:id]).destroy
  flash[:notice] = "The action was successfully deleted"
  redirect_to :action => "list"
end

protected

def actions
  load_listing
  @actions
end

def action_pages
  load_listing
  @action_pages
end

def load_listing
  unless @actions
    @action_pages, @actions = paginate :actions, :include => [:project, :context], :per_page =>
5, :order => "actions.description"
  end
end

def page
  params[:page] || 1
end

```

Com isso concluímos a nossa passagem sobre a parte de *caching* do Rails, notando que ela realmente é muito poderosa mas que deve ser usada com cuidado. Caso você queria, você pode desabilitar a parte de *caching* em desenvolvimento no momento, retornando o arquivo config/environments/development.rb para o seguinte:

```

# Settings specified here will take precedence over those in config/environment.rb

# In the development environment your application's code is reloaded on
# every request. This slows down response time but is perfect for development
# since you don't have to restart the webserver when you make code changes.
config.cache_classes = false

# Log error messages when you accidentally call methods on nil.
config.whiny_nils = true

# Enable the breakpoint server that script/breakpointer connects to
config.breakpoint_server = true

# Show full error reports and disable caching
config.action_controller.consider_all_requests_local = true
config.action_controller.perform_caching = false
config.action_view.cache_template_extensions = false
config.action_view.debug_rjs = true

# Don't care if the mailer can't send
config.action_mailer.raise_delivery_errors = false

```

Reinic peace o servidor para que as modificações tenham efeito.

Até o momento, nossa aplicação ficou restrita ao idioma inglês, até mesmo para não termos que nos preocupar com particularidades do Rails. Obviamente esse provavelmente não é o seu caso, desenvolvendo aplicações em português.

Por padrão, o Rails não provê nenhuma funcionalidade relacionada a localização e internacionalização. Ao contrário, a maioria de suas características de geração automática de nomes é exclusivamente voltada para o idioma inglês.

Apesar disso, sem grande esforço, podemos modificar algumas partes do mesmo para que possamos localizar nossas aplicações em uma certa medida. Uma dessas maneiras é a redefinição de alguns métodos e constantes. Caso sua aplicação não precise funcionar em vários idiomas simultaneamente, isso é geralmente suficiente para que você possa usar outro idioma.

Um caso básico é o dos nomes das tabelas, que é derivado automaticamente do nome da classe de dados. Caso você não se sinta confortáveis em usar nomes de modelos em inglês, preferindo nomes em português, você pode fazer algum assim:

```
class Contexto < ActiveRecord::Base  
  set_table_name "contextos"  
  set_primary_key "codigo_contexto"  
end
```

Com essas duas modificações o Rails sabe que nem o nome da tabela, nem o nome de sua chave primária surrogada são os mesmos e que os nomes customizados fornecidos devem ser usados. As modificações se aplicam a qualquer chamada e não oferecem quase nenhum problema em qualquer das demais funcionalidades do Rails. Não é necessário se preocupar com o nome dos atributos já que eles são automaticamente derivados do banco de dados. Migrações também aceitam modificações sem muito esforço.

Obviamente isso não é localização nem internacionalização no sentido básico das palavras, mas o código também é um ponto em que o uso de seu idioma—com em qualquer outra linguagem de programação—ode ser um problema. Usar o inglês provavelmente representa menos necessidade de mexer aqui e ali em sua aplicação, mas não é um motivo para dificuldades.

Entretanto, como mencionado acima, alguns pontos do Rails que esperam nomes em inglês podem começar a apresentar alguns problemas. Um exemplo disso é o método `error_messages_for`, que gerar um bloco fechado de código descrevendo os erros gerados por um modelo. Esse método é definido da seguinte maneira no código do Rails:

```

def error_messages_for(object_name, options = {})
  options = options.symbolize_keys
  object = instance_variable_get("@#{object_name}")
  if object && !object.errors.empty?
    content_tag("div",
      content_tag(
        options[:header_tag] || "h2",
        "#{pluralize(object.errors.count, "error")}" prohibited this #{object_name.to_s.gsub("_", " ")}) from being saved"
      ) +
      content_tag("p", "There were problems with the following fields:") +
      content_tag("ul", object.errors.full_messages.collect { |msg| content_tag("li", msg) }),
      "id" => options[:id] || "errorExplanation", "class" => options[:class] || "errorExplanation"
    )
  else
    ""
  end
end

```

Note que existem vários problemas com a implementação: primeiro, o texto das mensagens usadas está embutido direto do método e não pode ser configurado. Segundo, o método usa pluralize para variar a frase que informa se há um ou mais erros. Como esse método pluraliza bem somente palavras em inglês (e mesmo assim sofre de várias excessões) temos mais um ponto complicado.

A solução mais óbvia seria substituir completamente o método, usando nossa própria implementação para isso. Poderíamos, por exemplo, adicionar a seguinte implementação em nosso arquivo de *helper* primárip, *application_helper.rb*:

```

def error_messages_for(object_name, options = {})
  options = options.symbolize_keys
  object = instance_variable_get("@#{object_name}")
  if object && !object.errors.empty?
    content_tag("div",
      content_tag(
        options[:header_tag] || "h2",
        "#{pluralize(object.errors.count, "erro impediu", "erros impediram")}" que esse registro fosse
salvo:")
      ) +
      content_tag("p", "Os seguintes campos apresentaram problemas:") +
      content_tag("ul", object.errors.full_messages.collect { |msg| content_tag("li", msg) }),
      "id" => options[:id] || "errorExplanation", "class" => options[:class] || "errorExplanation"
    )
  else
    ""
  end
end

```

Fazendo uma validação qualquer, isso resolverá parcialmente o nosso problema. O cabeçalho de nossa informação de validação será corrigido, mas não as mensagens de erro por campo, que são geradas automaticamente pelo Rails. Uma forma de contornar isso seria redefinir o conjunto que descreve internamente essas mensagens. Isso poderia ser feito acrescentando a seguinte modificação ao final do arquivo *environment.rb*:

```

module ActiveRecord
  class Errors

```

```

@@default_error_messages = {
  :inclusion => "não existe na lista",
  :exclusion => "já existe na lista",
  :invalid => "é inválido.",
  :confirmation => "não confere com sua confirmação",
  :accepted => "deve ser aceito",
  :empty => "não pode ser vazio",
  :blank => "não pode estar em branco",
  :too_long => "é muito longo (o máximo deve ser %d caracteres)",
  :too_short => "é muito curto (o mínimo deve ser %d caracteres)",
  :wrong_length => "possui o comprimento errado (o tamanho deveria ser %d caracteres)",
  :taken => "já foi usado em outro registro",
  :not_a_number => "não é um número"
}
end
end

```

O código acima substitui a variável de classe que representa as mensagens. Se o nome de seus campos estiverem em português, isso já será uma espécie de localização. Um problema, porém, é que as mensagens ficam um pouco estranhas. Você teria mensagens como:

```

Nome não pode ser vazio.
Senha não confere com sua confirmação.
Login já foi usado em outro registro.

```

Obviamente, isso não é muito interessante. Poderíamos melhorar essas mensagens um pouco fazendo uma outra modificação em nosso método `error_messages_for`, deixando o mesmo assim:

```

def error_messages_for(object_name, options = {})
  options = options.symbolize_keys
  object = instance_variable_get("@#{object_name}")
  unless object.errors.empty?
    items = []
    object.errors.each { |attribute, message| items << content_tag("li", message) }
    content_tag("div",
      content_tag(
        options[:header_tag] || "h2",
        "#{pluralize(object.errors.count, "erro impediu", "erros impediram")} que esse registro
fosse salvo:")
      +
      content_tag("p", "Os seguintes campos apresentaram problemas:") +
      content_tag("ul", items.join("")),
      "id" => options[:id] || "errorExplanation", "class" => options[:class] || "errorExplanation"
    )
  end
end

```

Essa implementação exibe somente a mensagem pura adicionada à validação. Com o método acima, modificariamos um modelo de dados qualquer para exibir mensagens completas na validação, como no exemplo abaixo:

```

class Contexto < ActiveRecord::Base

  set_table_name "contextos"
  set_primary_key "codigo_contexto"

```

```
validates_presence_of :name, :message => "O nome do contexto deve ser informado"
validates_presence_of :user_id, :message => "O usuário associado ao contexto deve ser informado"

end
```

Embora possa parecer um inconveniente explicitar a mensagem de validação por completo, esse método provavelmente é o mais interessante por permitir que a mensagem seja cuidadosamente cunhada para cada campo e validação, aumentando bastante o grau de legibilidade das mesmas.

As alterações acima provavelmente resolvem a maior parte da necessidade de tradução em suas aplicações. Apesar disso, talvez você tenha um outro problema no uso de caracteres acentuados. Esses problemas geralmente decorrem do uso de formatos diferente de armazenamento de texto entre as várias partes de sua aplicação, passando desde a representação interna do Ruby até a persistência final no banco de dados.

A solução provavelmente mais acertada para o problema é usar uma codificação comum para todos, o que dependenderá de alguns fatores como o servidor de banco de dados que você está usando. Nesse ponto, o uso de Unicode com o formato UTF-8 seria provavelmente a escolha mais interessante. Mas temos um grande problema no fato de que o Ruby não suporta Unicode nativamente.

Apesar disso, existem algumas passos que você pode tomar para que sua aplicação basicamente fale Unicode sem que o Ruby precise saber disso na maior parte dos casos.

O primeiro passo, obviamente, seria escolher um banco de dados que suporte Unicode com UTF-8 nativamente. O MySQL, por exemplo, somente tem suporte a isso a partir da versão 4.1. Inclusive, a partir da versão 5.0, a instalação padrão habilita o UTF-8 como codificação padrão. Em versões anteriores, você pode usar algo assim para gerar suas tabelas com suporte a UTF-8:

```
create table contexts
(
  id int not null auto_increment,
  foo varchar(100) not null,
  primary key (id)
) type=myisam character set utf8;
```

Em seguida, precisamos de um filtro em nossa aplicação para fazer com que o Rails retorne por padrão na em UTF-8 e que também converse em UTF-8 com o banco de dados. Poderíamos ter algo assim em um arquivo application.rb:

```
class ApplicationController < ActionController::Base
  before_filter :set_charset

  def set_charset
    @response.headers["Content-Type"] = "text/html; charset=utf-8"
  end
end
```

O código acima indica ao navegador que o retorno da aplicação é UTF-8.

Por fim, precisamos mudar o nosso arquivo de configuração do banco de dados para fazer com que a comunicação com o banco de dados também seja em UTF-8. Para o MySQL, a configuração seria assim:

```
development:  
  adapter: mysql  
  database: db  
  encoding: utf8  
  username: root  
  password:
```

Obviamente é importante também que você esteja usando um editor que salve os seus arquivos que você está gerando em UTF-8. Caso contrário, o que acontecerá é que qualquer texto problemático (leia-se com acentos) em seus *controllers*, *models* e *views* entrará em conflito com o formato do resto dos dados, gerando páginas que misturam duas ou mais codificações, consequentemente fazendo com que sua aplicação falhe.

Os passos acima são capazes de ajudar a resolver a maior parte dos problemas com codificação de caracteres do Rails, mas ainda nos deixam com um problema fundamental: o fato de que nenhum dos métodos da classe `String` do Ruby suporta Unicode e que qualquer uso dos mesmos em texto que contenha as variações de *bytes* por caractere presentes no UTF-8 causará problemas. Mesmo um método simples como `length` deixará de funcionar.

Uma solução quase completa para isso é usar bibliotecas para isso. Por exemplo, usando o seguinte código antes de qualquer coisa em sua aplicação (no começo do arquivo `environment.rb`, por exemplo), garantirá um suporte básico ao UTF-8 no Rails:

```
$KCODE = "u"  
require "jcode"
```

Esse código não cuida de todas as funções do Ruby, mas podemos adicionar suporte para mais algumas usando uma outra biblioteca. Para usá-la, primeiramente a instalamos via `gem`:

```
ronaldo@minerva:~/tmp/gtd$ gem install unicode  
Bulk updating Gem source index for: http://gems.rubyforge.org  
...  
Successfully installed unicode-0.1
```

Com essa biblioteca instalada, podemos usar os métodos abaixo no lugar dos usuais:

```
Unicode::downcase(string)  
Unicode::upcase(string)  
# etc, etc, etc...
```

Isso resolve quase todos os problemas que poderíamos ter. Mas, até que uma versão do Ruby seja lançada com suporte nativo a Unicode, como está previsto, não é possível ter 100% de confiança que as conversões de dados entre o código e as demais partes da aplicação funcionam completamente. Você pode ler mais sobre o assunto em: <http://wiki.rubyonrails.com/rails/pages/HowToUseUnicodeStrings>.

Isso, entretanto, não tem impedido que o Rails seja usado para aplicações que não estão em inglês. Eu venho fazendo isso há quase dois anos já e ainda não tive qualquer problema seguido passos similares aos descritos acima.

Um último tópico antes de terminarmos essa sessão é globalização, ou seja, a adaptação de uma aplicação em termos de localização e internacionalização para que ela suporte vários idiomas simultaneamente, de acordo com preferências de instalação ou de usuário.

Para essas tarefas, existem *plugins* para o Rails que fazem um trabalho melhor do que poderíamos fazer nesse tutorial tentando qualquer coisa manualmente. Dois desses *plugins* são:

Globalize

<http://plugins.radrails.org/directory/show/59>

GLoc

<http://plugins.radrails.org/directory/show/31>

Visitando a página dos mesmos pode entender mais sobre como cada um deles funciona e quais são suas respectivas vantagens e desvantagens.

SEGURANÇA

O Rails, por padrão, já faz um bom trabalho de tentar evitar os problemas mais comuns de segurança de dados em uma aplicação. Entretanto, o desenvolvedor deve estar sempre atento para não introduzir problemas próprios em suas aplicações. A presente seção detalha algumas dicas de segurança que podem ser seguidas para garantir uma confiabilidade maior ao seu código.

Um dos maiores problemas em aplicações Web é a possibilidade de injeção de SQL, que acontece quando uma aplicação inclui dados em chamadas ao banco de dados que estão vindo de fontes inseguras. Um exemplo seria o código abaixo:

```
User.find :all, :conditions => ["name like '%#{params[:keywords]}%'"]
```

Esse código gera a seguinte declaração SQL, ou algo parecido pelo menos, para um parâmetro qualquer:

```
select * from users where name like '%test%'
```

Agora, imagine que um usuário malicioso, ao invés de passar um palavra-chave comum, passe o seguinte texto como parâmetro:

```
' ; delete * from users where '1=1
```

O nosso código SQL seria transformado, depois da interpolação em:

```
select * from users where name like '%test%'; delete * from users where '1=1'
```

O resultado seria a exclusão de todos nossos registros de usuário. Do exemplo acima, você pode ver que ataques bastante sofisticados poderiam acontecer. Se o SQL construído for suficientemente versátil, um pouco de testes poderá revelar inclusive detalhes de *logins* e senhas em um banco de dados.

A solução para o problema acima é usar sempre parâmetros nomeados. A busca original ficaria assim:

```
User.find :all, :conditions => ["name like ?", "%#{params[:keywords]}"]
```

Usando parâmetros nomeados, qualquer caractere potencialmente perigoso será tratado previamente. Uma outra prática interessante é extrair consultas freqüentemente usadas para métodos, como fizemos algumas vezes ao longo de nosso usuário. O método acima, por exemplo, poderia ser transformado em:

```
class User < ActiveRecord::Base  
  
  def self.search_by_name(keywords)  
    find :all, :conditions => ["name like ?", "%#{keywords}%"]  
  end  
  
end
```

Um outro problema, bem similar, é usar parâmetros em *templates RJS*, sujeitando a aplicação a ataques JavaScript. Por exemplo, o código abaixo em uma *view* usando RJS é vulnerável:

```
<%= @params["keywords"] %>
```

Se o código acima for enviado em resposta a uma URL como a abaixo, você verá um *alert* com o valor dos *cookies* setados em sua aplicação, incluindo o *session id* do usuário:

```
http://localhost:3000/test/action?keywords=%3Cscript%3Ealert%28document.cookie%29%3C%2Fscript%3E
```

Essa é mais uma questão de proteger a sua aplicação de ser raptada por código malicioso injetado em formulários abertos.

A solução é tratar o resultado antes, usando algo como:

```
<%=h @params["keywords"] %>
```

O método `h`, no Rails, trata a sua entrada transformando qualquer caractere problemático em seu equivalente HTML, prevenindo a maior partes dos problemas. O código acima pode ser usado tanto em *templates RJS* como *views HTML* normais.

Um último tipo de problema é o uso de atribuições diretas via formulários, usando o método `attributes` das classes de dados. Suponha que, em nossa aplicação, tenhamos um formulário que permita que qualquer pessoa se cadastre para ser um novo usuário. O nosso código atual, como podemos ver abaixo, usa exatamente esse método de atribuição:

```
class UsersController < ApplicationController

  before_filter :authenticate_administration

  def index
    list
    render :action => "list"
  end

  def list
    @user_pages, @users = paginate :users, :per_page => 5, :order => "name"
  end

  def edit
    if params[:id]
      @user = User.find(params[:id])
    else
      @user = User.new
    end
    if request.post?
      @user.attributes = params[:user]
      send_email = @user.new_record?
      if @user.save
        expire_action(:action => "list")
        if send_email
          RegistrationNotifier.deliver_registration_notification(@user, session_user)
        end
        flash[:notice] = "The user was successfully saved"
        redirect_to :action => "list"
      end
    end
  end

  def delete
    User.find(params[:id]).destroy
    flash[:notice] = "The user was successfully deleted"
    redirect_to :action => "list"
    expire_action(:action => "list")
  end
end
```

Se usássemos o mesmo código na parte pública do site (removendo a autenticação da inclusão de dados), teríamos um enorme problema: qualquer usuário poderia se configurar administrador do sistema. Isso

acontece porque o atributo `admin` da classe pode ser livremente atribuído. Assim, mesmo que não exibíssemos uma caixa de seleção para esse atributo, um atacante malicioso poderia simplesmente criar um página que enviasse isso por ele, evitando tranquilamente qualquer o nosso pobre método de obscurecimento.

A solução é modificar tanto o nosso modelo para evitar atribuições diretas:

```
class User < ActiveRecord::Base

  attr_accessor :current_user_id

  validates_presence_of :name
  validates_presence_of :login
  validates_presence_of :password
  validates_presence_of :email

  validates_uniqueness_of :login

  attr_protected :admin

end
```

Com essa modificação, nossa ação pública não seria mais capaz de atribuir automaticamente um valor que indicasse acesso administrativo ao banco de dados. E nossa ação administrativa teria que ser modificada para algo assim:

```
class UsersController < ApplicationController

  before_filter :authenticate_administration

  def index
    list
    render :action => "list"
  end

  def list
    @user_pages, @users = paginate :users, :per_page => 5, :order => "name"
  end

  def edit
    if params[:id]
      @user = User.find(params[:id])
    else
      @user = User.new
    end
    if request.post?
      @user.attributes = params[:user]
      @user.admin = params[:user][:admin]
      send_email = @user.new_record?
      if @user.save
        expire_action(:action => "list")
        if send_email
          RegistrationNotifier.deliver_registration_notification(@user, session_user)
        end
        flash[:notice] = "The user was successfully saved"
        redirect_to :action => "list"
      end
    end
  end
end
```

```
def delete
  User.find(params[:id]).destroy
  flash[:notice] = "The user was successfully deleted"
  redirect_to :action => "list"
  expire_action(:action => "list")
end
end
```

Todas as modificações que vimos até agora são bem simples, mas o uso das mesmas pode evitar incontáveis problemas no desenvolvimento de uma aplicação Web.

UNIT TESTING

Embora tenhamos ignorado esse assunto durante todo o nosso tutorial, indiscutivelmente uma das partes mais importantes de qualquer aplicação Rails são os testes que devem ser mantidos sobre a mesma. *Unit testing* é uma parte fundamental da metodologia *Extreme Programming*, cujas filosofias formam a base de muitas decisões tomadas no Rails.

A pedra de esquina da metodologia *Extreme Programming* (XP) é o teste de unidade (*unit test*), ou como alguns o chamam, teste unitário. Um teste, na metodologia XP é uma das maneiras de saber se o código que foi escrito se conforma aos requerimentos do mesmo e, mais ainda, se mudanças introduzidas posteriormente à criação do mesmo não invalidaram código já existente, introduzindo *bugs* na aplicação. Por causa disso, aplicações seguindo a metodologia XP sempre começam o desenvolvimento a partir dos testes, definindo as condições segundo as quais o código a ser desenvolvido deve ser criado. O racional para isso é o fato simples de que, se um teste razoavelmente completo passa, o código escrito provavelmente está correto.

O objetivo desse tutorial não é oferecer uma introdução à metodologia XP, mas mostrar, pelo menos inicialmente, como a mesma pode ser aplicada ao Rails no que concerne a testes. Para propósitos do tutorial, os testes foram ignorados até o momento porque introduzi-los significaria desviar a nossa atenção da demonstração do Rails em si. Se você deseja aprender mais sobre XP e testes, abaixo estão alguns *links* recomendados:

http://en.wikipedia.org/wiki/Unit_testing
http://en.wikipedia.org/wiki/Test-driven_development
<http://c2.com/cgi/wiki?TestDrivenDevelopment>
<http://www.xprogramming.com/testfram.htm>

O Rails oferecia, até a versão 1.1, duas formas principais de testes: testes de unidade, já mencionados acima, e testes funcionais. Os primeiros testam os modelos de dados, garantindo que validações, associações e quaisquer outros adendos ao básico funcionam correstamente. Os últimos testam se a funcionalidade básica da aplicação, como presente em *controllers* e *views*, está correta.

Um novo tipo de testes foi introduzido com o Rails 1.1, que são os testes de integração. Esses testes são

responsáveis por verificar o funcionamento de partes conectadas de uma aplicação. São uma expansão dos testes funcionais da mesma forma que os testes funcionais são uma extensão dos testes de unidade. Em outras palavras, os testes de unidade verificam se as classes de dados estão funcionando corretamente; os testes funcionais verificam se o uso da mesmas está correto nos *controllers*; e os testes de integração verificam se as interdependências entre as mesmas também estão corretas—sendo que estas duas últimas verificações são feitas sobre *controllers* e *views* também.

Para fazer uso de testes, o Rails utiliza a biblioteca `test/unit`, que vem por padrão com o Ruby. A biblioteca é transparentemente incluída no Rails e utilizada também de maneira bem transparente. A maioria dos comandos de geração que vimos até o momento cria automaticamente testes para cada peça introduzida na aplicação. Para testar completamente uma aplicação em Rails, usamos o comando abaixo:

```
rake test
```

O resultado do comando abaixo, na aplicação que temos agora, é algo assim:

```
ronaldo@minerva:~/tmp/gtd$ rake test
(in /home/ronaldo/tmp/gtd)
/usr/bin/ruby1.8 -Ilib:test "/usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader.rb"
"test/unit/project_test.rb" "test/unit/action_test.rb" "test/unit/context_test.rb"
"test/unit/user_test.rb" "test/unit/resource_test.rb" "test/unit/registration_notifier_test.rb"
"test/unit/audit_test.rb"
Loaded suite /usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader
Started
.....
Finished in 0.106873 seconds.

7 tests, 7 assertions, 0 failures, 0 errors
/usr/bin/ruby1.8 -Ilib:test "/usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader.rb"
"test/functional/home_controller_test.rb" "test/functional/controllers_controller_test.rb"
"test/functional/projects_controller_test.rb" "test/functional/actions_controller_test.rb"
"test/functional/login_controller_test.rb" "test/functional/resources_controller_test.rb"
"test/functional/users_controller_test.rb" "test/functional/feed_controller_test.rb"
Loaded suite /usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader
Started
.....
Finished in 0.02817 seconds.

8 tests, 8 assertions, 0 failures, 0 errors
/usr/bin/ruby1.8 -Ilib:test "/usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader.rb"
Loaded suite /usr/bin/rake
Started

Finished in 0.000308 seconds.

0 tests, 0 assertions, 0 failures, 0 errors
```

O comando acima roda tanto os testes de unidade com os testes funcionais da aplicação. Caso desejemos, podemos testá-los separadamente, como mostrado abaixo:

```
ronaldo@minerva:~/tmp/gtd$ rake test:units
(in /home/ronaldo/tmp/gtd)
/usr/bin/ruby1.8 -Ilib:test "/usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader.rb"
```

```

"test/unit/project_test.rb" "test/unit/action_test.rb" "test/unit/context_test.rb"
"test/unit/user_test.rb" "test/unit/resource_test.rb" "test/unit/registration_notifier_test.rb"
"test/unit/audit_test.rb"
Loaded suite /usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader
Started
.....
Finished in 0.261947 seconds.

7 tests, 7 assertions, 0 failures, 0 errors
Loaded suite /usr/bin/rake
Started

Finished in 0.000316 seconds.

0 tests, 0 assertions, 0 failures, 0 errors
ronaldo@minerva:~/tmp/gtd$ rake test:functionals
(in /home/ronaldo/tmp/gtd)
/usr/bin/ruby1.8 -Ilib:test "/usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader.rb"
"test/functional/home_controller_test.rb" "test/functional/context_controller_test.rb"
"test/functional/projects_controller_test.rb" "test/functional/actions_controller_test.rb"
"test/functional/login_controller_test.rb" "test/functional/resources_controller_test.rb"
"test/functional/users_controller_test.rb" "test/functional/feed_controller_test.rb"
Loaded suite /usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader
Started
.....
Finished in 0.045681 seconds.

8 tests, 8 assertions, 0 failures, 0 errors
Loaded suite /usr/bin/rake
Started

Finished in 0.000331 seconds.

0 tests, 0 assertions, 0 failures, 0 errors

```

Como podemos ver pelos comandos usados, o que essa tarefa do `rake` faz é meramente invocar a biblioteca `test/unit` para rodar cada um dos testes já criados até o momento, que passam por não conterem mais do que um mero teste retornando verdadeiro.

Um arquivo de caso de testes, seja funcional ou de unidade, pode conter vários testes dentro de si. De fato, a hierarquia de testes segue o seguinte formato:

- Todos os arquivos de teste de uma aplicação forma uma *suite* de testes;
- Uma *suite* de testes possui muitos casos de teste;
- Um caso de testes possui muitos testes;
- Um teste possui muitas asserções.

As asserções, com o próprio nome diz, são afirmações sobre a validade de uma determinada parte do código. Uma asserção poderia, por exemplo, dizer se ao atribuirmos o valor verdadeiro ao atributo `done` de nossa classe `Action`, o valor do atributo `completed` é configurado para uma data. Um teste qualquer possui várias asserções relacionadas.

No Rails, é aqui que entra o banco de testes que configuramos em nosso aplicação. Ao rodarmos o *suite* de testes da aplicação, esse banco é destruído e recriado limpo para que os testes possam operar em dados confiáveis e funcionar corretamente. O banco existe para justamente não termos que nos preocupar com os

ambientes de produção e desenvolvimento, deixando-os intocados—principalmente para o caso em que estivermos fazendo testes que envolvem a remoção de dados.

Se você observar o diretório `test` que se encontra em sua aplicação, verá uma série de diretórios que você poderá usar:

```
ronaldo@minerva:~/tmp/gtd/test$ ls -l
total 28
drwxr-xr-x 3 ronaldo ronaldo 4096 2006-09-24 22:54 fixtures
drwxr-xr-x 2 ronaldo ronaldo 4096 2006-09-24 17:35 functional
drwxr-xr-x 2 ronaldo ronaldo 4096 2006-09-22 23:17 integration
drwxr-xr-x 4 ronaldo ronaldo 4096 2006-09-22 23:17 mocks
-rw-r--r-- 1 ronaldo ronaldo 1317 2006-09-22 23:17 test_helper.rb
drwxr-xr-x 3 ronaldo ronaldo 4096 2006-09-24 19:22 tmp
drwxr-xr-x 2 ronaldo ronaldo 4096 2006-10-01 23:31 unit
```

Editaremos justamente arquivos nesses diretórios para realizar os testes que precisamos.

Dois dos diretórios acima são de particular interesse. Para testarmos se a funcionalidade relativa aos modelos está correta, é óbvio que precisamos de dados para isso. O diretório `fixtures` contém justamente arquivos que nos permitem informar dados a serem carregados no banco de dados para testes durante a execução dos mesmos. Já o diretório `mocks` pode eventualmente conter classes que simulam aspectos da aplicação não disponíveis durante testes—um exemplo seria funcionalidade que depende de serviços de rede que não podem ser usados para homologação.

Já falamos bastante sobre a teoria por trás dos testes no Rails. Vamos colocar isso em prática criando nossos primeiros testes de unidade.

Para começar, vamos carregar alguns dados em nosso banco usando algumas *fixtures*. Dentro desse diretório, vemos uma série de arquivos `.yml` correspondendo aos nossos modelos. Editaremos o arquivo relativo a usuários que, no momento, contém o seguinte:

```
first:
  id: 1
another:
  id: 2
```

Vamos modificar o arquivo para algo assim:

```
ronaldo:
  id: 1
  name: Ronaldo
  login: ronaldo
  password: test
  admin: true
  email: ronaldo@reflectivesurface.com
marcus:
  id: 2
  name: Marcus
```

```
login: marcus
password: test
admin: false
email: marcus@reflectivesurface.com
```

Os dados que acabamos de introduzir serão automaticamente inseridos em nosso banco de testes quando precisarmos dos mesmos. Agora que temos alguns dados, vamos testar a funcionalidade presente em nossos modelos. Para isto, vamos editar o arquivo `user_test.rb`, que está no diretório `unit`.

Esse arquivo está definido assim no momento:

```
require File.dirname(__FILE__) + '/../test_helper'

class UserTest < Test::Unit::TestCase
  fixtures :users

  # Replace this with your real tests.
  def test_truth
    assert true
  end
end
```

Como podemos ver, esse arquivo define uma classe `UserTest` que é um caso de teste, como evidenciado pelo nome da qual é derivada. Esse caso de teste contém um único teste no momento, que simplesmente passa por testar algo que sempre será verdadeiro.

Veja que o arquivo já carrega as *fixtures* necessárias para o mesmo. É nesse momento que nossos dados de teste são carregados do banco. Se quisermos, podemos carregar outras *fixtures* simplesmente declarando-as com o mesmo comando. O símbolo passado para a função representa o arquivo no diretório `fixtures`.

Nós usamos esses arquivos definindo métodos que representam testes. Vamos começar com o básico, testando se nossos dados podem ser recuperados do banco sem problemas:

Modificamos o nosso arquivo de teste para o seguinte:

```
require File.dirname(__FILE__) + '/../test_helper'

class UserTest < Test::Unit::TestCase
  fixtures :users

  def test_basic_selections
    assert_equal "Ronaldo", users(:ronaldo).name
    assert_equal "Marcus", users(:marcus).name
    assert_equal 2, User.count
  end
end
```

Como podemos ver, removemos o teste anterior e criamos o nosso próprio teste. O primeiro teste usa a

asserção `assert_equal` para comparar dois valores.

Na primeira das asserções acima, verificamos se temos realmente dois usuários no banco. Como declaramos dois usuários em nossas *fixtures*, esse teste deveria retornar corretamente.

A segunda e a terceira asserção recuperam dois usuários do banco e verificam se os seus nomes correspondem ao que esperamos. Note a construção usada para recuperar os dados do banco. Os símbolos que usamos para declarar nossos usuários no arquivo de *fixtures* podem ser usados para recuperar esses mesmos dados do banco de dados, por atribuição.

Rodando os testes, veremos se eles passam:

```
ronaldo@minerva:~/tmp/gtd$ rake test:units
(in /home/ronaldo/tmp/gtd)
/usr/bin/ruby1.8 -Ilib:test "/usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader.rb"
"test/unit/project_test.rb" "test/unit/action_test.rb" "test/unit/context_test.rb"
"test/unit/user_test.rb" "test/unit/resource_test.rb" "test/unit/registration_notifier_test.rb"
"test/unit/audit_test.rb"
Loaded suite /usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader
Started
.....
Finished in 0.305564 seconds.

7 tests, 9 assertions, 0 failures, 0 errors
Loaded suite /usr/bin/rake
Started

Finished in 0.000309 seconds.

0 tests, 0 assertions, 0 failures, 0 errors
```

Vemos que nos sete testes que temos, com suas nove asserções, todas passam. Uma maneira mais fácil de verificar se os testes específicos que estamos construindo estão passando é invocar diretamente o arquivo de testes, como no exemplo abaixo:

```
ronaldo@minerva:~/tmp/gtd$ ruby test/unit/user_test.rb
Loaded suite test/unit/user_test
Started
.
Finished in 0.064147 seconds.

1 tests, 3 assertions, 0 failures, 0 errors
```

Cada arquivo de testes é auto-contido e pode ser testado corretamente. Como esperamos, temos um teste e três asserções que passam sem problemas.

Vamos dizer que precisamos de mais um usuário. Para isso modificamos o nosso arquivo de *fixtures*:

```
ronaldo:
  id: 1
  name: Ronaldo
```

```

login: ronaldo
password: test
admin: true
email: ronaldo@reflectivesurface.com
marcus:
  id: 2
  name: Marcus
  login: marcus
  password: test
  admin: false
  email: marcus@reflectivesurface.com
alessandra:
  id: 3
  name: Alessandra
  login: alessandra
  password: test
  admin: false
  email: alessandra@reflectivesurface.com

```

Se rodarmos o nosso teste agora, veremos o seguinte:

```

ronaldo@minerva:~/tmp/gtd$ ruby test/unit/user_test.rb
Loaded suite test/unit/user_test
Started
F
Finished in 0.285283 seconds.

1) Failure:
test_basic_selections(UserTest) [test/unit/user_test.rb:9]:
<2> expected but was
<3>.

1 tests, 3 assertions, 1 failures, 0 errors

```

Vamos agora que, dos três testes que estabelecemos, um falhou. Isso é denotado não só pela listagem geral no fim do teste comotambém pela linha que começa com uma letra F maiúscula (*failure*).

O teste nos informa também o que falhou, para que possamos corrigi-lo. No caso aqui, vamos corrigir o nosso teste, já que a falha está no mesmo. Em testes elaborados corretamente, porém, basicamente qualquer problema seria corrigido no código—a menos que o teste realmente contenha um *bug*.

O nosso teste agora seria:

```

require File.dirname(__FILE__) + '/../test_helper'

class UserTest < Test::Unit::TestCase
  fixtures :users

  def test_basic_selections
    assert_equal "Ronaldo", users(:ronaldo).name
    assert_equal "Marcus", users(:marcus).name
    assert_equal 3, User.count
  end
end

```

E passaria sem problemas.

Uma coisa a se manter em mente é que a primeira asserção a falhar em um teste invalida todo o teste. Isso se justifica pelo fato de que um teste é uma unidade, que deve representar um conjunto completo de asserções sobre o que se quer testar de modo que nada—até onde é humanamente possível—fique de fora.

Digamos agora que queremos estender os nossos testes para testar operações básicas em nosso modelo de dados.

```
require File.dirname(__FILE__) + '/../test_helper'

class UserTest < Test::Unit::TestCase
  fixtures :users

  def test_basic_selections
    assert_equal "Ronaldo", users(:ronaldo).name
    assert_equal "Marcus", users(:marcus).name
    assert_equal 3, User.count
  end

  def test_crud
    user = User.new(:name => "Renato")
    assert user.save
  end
end
```

Rodando esse teste, temos uma falha, obviamente, já que só informarmos o nome do usuário e temos algumas validações que impedem que o usuário seja salvo sem dados.

```
ronaldo@minerva:~/tmp/gtd$ ruby test/unit/user_test.rb
Loaded suite test/unit/user_test
Started
.F
Finished in 0.291654 seconds.

1) Failure:
test_crud(UserTest) [test/unit/user_test.rb:14]:
<false> is not true.

2 tests, 4 assertions, 1 failures, 0 errors
```

Veja que agora temos dois testes, com quatro asserções, das quais uma está falhando. A linha logo abaixo da frase “Started” dá uma representação “visual” de nossos testes mostrando que o primeiro passou, mas o segundo não.

Obviamente, como estamos criando os nossos testes depois de já termos construído a aplicação, os mesmos não fazem muito sentido. Daí a necessidade de testarmos antes de construir a aplicação, planejando em testes a funcionalidade que desejamos antes mesmo que ela esteja lá. A idéia por trás do XP não é modificar testes para que eles passem, mas modificar código para que ele satisfaça aos testes estabelecidos, considerando que estejam corretos. Se montarmos os nossos testes antes de qualquer coisa, toda a nossa

produção de código se resumiria à criação de código para implementar a funcionalidade que já estabelecemos.

Um teste um pouco mais sofisticado, que poderíamos ter elaborado no começo de nossa aplicação poderia ter sido assim:

```
require File.dirname(__FILE__) + '/../test_helper'

class UserTest < Test::Unit::TestCase
  fixtures :users

  def test_basic_selections
    assert_equal "Ronaldo", users(:ronaldo).name
    assert_equal "Marcus", users(:marcus).name
    assert_equal 3, User.count
  end

  def test_crud
    user = User.new(:name => "Renato", :login => "renato",
      :password => "renato", :email => "renato@reflectivesurface.com")

    assert user.save
    assert_equal 4, User.count
    assert !user.admin?

    user.password = "test"
    assert user.save

    same_user = User.find(user.id)
    assert_equal user.name, same_user.name

    different_user = User.find(1)
    assert_not_equal user.id, different_user.id
    assert_not_equal user.name, different_user.name

    assert user.destroy
    assert_equal 3, User.count
  end
end
```

Veja que estamos testando basicamente todas as operações básicas para um objeto de dados. Executando o nosso teste agora, temos:

```
ronaldo@minerva:~/tmp/gtd$ ruby test/unit/user_test.rb
Loaded suite test/unit/user_test
Started
..
Finished in 0.136343 seconds.

2 tests, 12 assertions, 0 failures, 0 errors
```

Para experimentar com a criação de funcionalidade que não temos no momento, vamos supor que queremos adicionar uma relacionamento entre usuários e ações. Como não temos isso em nosso modelo no momento, podemos começar de maneira simples:

```

require File.dirname(__FILE__) + '/../test_helper'

class UserTest < Test::Unit::TestCase
  fixtures :users

  def test_basic_selections
    assert_equal "Ronaldo", users(:ronaldo).name
    assert_equal "Marcus", users(:marcus).name
    assert_equal 3, User.count
  end

  def test_crud
    user = User.new(:name => "Renato", :login => "renato",
      :password => "renato", :email => "renato@reflectivesurface.com")

    assert user.save
    assert_equal 4, User.count
    assert !user.admin?

    user.password = "test"
    assert user.save

    same_user = User.find(user.id)
    assert_equal user.name, same_user.name

    different_user = User.find(1)
    assert_not_equal user.id, different_user.id
    assert_not_equal user.name, different_user.name

    assert user.destroy
    assert_equal 3, User.count
  end

  def test_actions_relationship
    user = users(:ronaldo)

    assert_respond_to user, :actions
    assert_equal 0, user.actions.size
  end
end

```

Em nosso novo teste, verificamos se a classe responde a um método `actions`, que representaria o nosso relacionamento e, em seguida, se o tamanho que esperamos para o conjunto de ações de um usuário é vazio. O resultado do teste é:

```

ronaldo@minerva:~/tmp/gtd$ ruby test/unit/user_test.rb
Loaded suite test/unit/user_test
Started
F..
Finished in 0.31562 seconds.

1) Failure:
test_actions_relationship(UserTest) [test/unit/user_test.rb:41]:
<#<User:0xb7752e40
@attributes=
  {"name"=>"Ronaldo",
   "admin"=>"1",
   "id"=>"1",

```

```

"password"=>"test",
"login"=>"ronaldo",
"email"=>"ronaldo@reflectivesurface.com"}>>
of type <User>
expected to respond_to?:actions.

3 tests, 13 assertions, 1 failures, 0 errors

```

Como esperado, temos uma falha. Implementamos a nossa funcionalidade agora com o seguinte código:

```

class User < ActiveRecord::Base

  attr_accessor :current_user_id

  validates_presence_of :name
  validates_presence_of :login
  validates_presence_of :password
  validates_presence_of :email

  validates_uniqueness_of :login

  has_many :actions

end

```

E nosso teste passa sem problemas:

```

ronaldo@minerva:~/tmp/gtd$ ruby test/unit/user_test.rb
Loaded suite test/unit/user_test
Started
...
Finished in 0.27348 seconds.

3 tests, 14 assertions, 0 failures, 0 errors

```

Essa é uma funcionalidade muito simples de testar, mas poderíamos ter adicionado código de teste que verificasse se o relacionamento funciona como queremos.

Poderíamos nos alongar mais, mas a idéia já está passada. Estes são os testes de unidade e podemos ver agora como são os testes funcionais.

Vamos trabalhar com o mesmo modelo de dados, usando o *controller* responsável pelo mesmo.

O arquivo, chamado apropriadamente `users_controller_test.rb`, está no diretório `functional` e contém o seguinte, no momento:

```

require File.dirname(__FILE__) + '/../test_helper'
require 'users_controller'

# Re-raise errors caught by the controller.
class UsersController; def rescue_action(e) raise e end; end

class UsersControllerTest < Test::Unit::TestCase

```

```

def setup
  @controller = UsersController.new
  @request   = ActionController::TestRequest.new
  @response  = ActionController::TestResponse.new
end

# Replace this with your real tests.
def test_truth
  assert true
end
end

```

Veja que a idéia é a mesma, com a diferença que temos agora um método `setup`, que é responsável por criar as condições necessárias para nosso teste. Um método oposto, chamada `teardown`, existe para descarregar o que criamos em `setup`, o que geralmente é necessário somente para objetos com dependências complexas, principalmente quando há recursos físicos associados e os mesmos devem ser reinicializados para testes posteriores.

Vamos testar agora alguns pontos em nosso *controller*:

```

require File.dirname(__FILE__) + '/../test_helper'
require 'users_controller'

# Re-raise errors caught by the controller.
class UsersController; def rescue_action(e) raise e end; end

class UsersControllerTest < Test::Unit::TestCase
  def setup
    @controller = UsersController.new
    @request   = ActionController::TestRequest.new
    @response  = ActionController::TestResponse.new
  end

  def test_index
    get :index
    assert_response :success
  end
end

```

Rodando o nosso teste, temos:

```

ronaldo@minerva:~/tmp/gtd$ ruby test/functional/users_controller_test.rb
Loaded suite test/functional/users_controller_test
Started
F
Finished in 0.055396 seconds.

  1) Failure:
test_index(UsersControllerTest) [test/functional/users_controller_test.rb:16]:
Expected response to be a <:success>, but was <302>

1 tests, 1 assertions, 1 failures, 0 errors

```

Uma falha óbvia já que a nossa ação espera um *login*. Um teste mais completo seria:

```

require File.dirname(__FILE__) + '/../test_helper'
require 'users_controller'

# Re-raise errors caught by the controller.
class UsersController; def rescue_action(e) raise e end; end

class UsersControllerTest < Test::Unit::TestCase
  fixtures :users

  def setup
    @controller = UsersController.new
    @request    = ActionController::TestRequest.new
    @response   = ActionController::TestResponse.new
  end

  def test_index
    get :index
    assert_redirected_to :controller => "login", :action => "login"
  end

  def test_index_with_login
    @request.session[:user] = user(:ronaldo).id
    get :index
    assert_response :success
  end
end

```

Se quisermos fazer algo ainda mais completo, podemos testar o que existe na resposta:

```

require File.dirname(__FILE__) + '/../test_helper'
require 'users_controller'

# Re-raise errors caught by the controller.
class UsersController; def rescue_action(e) raise e end; end

class UsersControllerTest < Test::Unit::TestCase
  fixtures :users

  def setup
    @controller = UsersController.new
    @request    = ActionController::TestRequest.new
    @response   = ActionController::TestResponse.new
  end

  def test_index
    get :index
    assert_redirected_to :controller => "login", :action => "login"
  end

  def test_index_with_login
    @request.session[:user] = users(:ronaldo).id
    get :index
    assert_response :success
    assert_tag :tag => "table", :children => { :count => User.count + 1,
      :only => { :tag => "tr" } }
  end
end

```

Aqui estamos testando se existe uma tabela com um número de linhas exatamente igual ao número de usuários que temos mais um (que seria o cabeçalho). Não estamos levando em conta uma possível paginação, mas isso também poderia ser considerado facilmente.

Continuando, poderíamos ter algum assim também:

```
require File.dirname(__FILE__) + '/../test_helper'
require 'users_controller'

# Re-raise errors caught by the controller.
class UsersController; def rescue_action(e) raise e end; end

class UsersControllerTest < Test::Unit::TestCase
  fixtures :users

  def setup
    @controller = UsersController.new
    @request   = ActionController::TestRequest.new
    @response  = ActionController::TestResponse.new
  end

  def test_index
    get :index
    assert_redirected_to :controller => "login", :action => "login"
  end

  def test_index_with_login
    @request.session[:user] = users(:ronaldo).id
    get :index
    assert_response :success
    assert_tag :tag => "table", :children => { :count => User.count + 1,
      :only => { :tag => "tr" } }
    assert_tag :tag => "td", :content => users(:ronaldo).name
  end
end
```

Estamos acima testando o conteúdo de uma única célula da tabela de usuários.

Testes funcionais são úteis para testar um *controller* em individual e pelos tipos de testes acima dá para ter uma idéia de como faríamos na maior parte dos casos. Testar Ajax é um pouco mais complicado porque a execução depende de JavaScript mas já existem *plugins* integrados a ferramentas próprias para isso que permitem um teste mais completo do que verificação de respostas textuais. Um *plugin* específico para isso pode ser encontrado no endereço: <http://andthennothing.net/archives/2006/02/05/selenium-on-rails>.

O último exemplo do que temos em termos de testes no Rails são os testes de integração mencionados anteriormente que nos permitem testar funcionalidade ao longo de vários *controllers*. Em nossa aplicação, poderíamos ter gerado um teste desses, por exemplo, para verificar o nosso processo de *login*. A título de exemplo, vamos fazer justamente isso.

Primeiro, vamos gerar o teste em si:

```
ronaldo@minerva:~/tmp/gtd$ script/generate integration_test login
exists  test/integration/
create  test/integration/login_test.rb
```

E agora, editar o arquivo:

```

require "#{File.dirname(__FILE__)}/../test_helper"

class LoginTest < ActionController::IntegrationTest
  fixtures :users

  def test_login
    user = users(:ronaldo)

    get "/home"
    assert_redirected_to "/login/login"

    follow_redirect!

    assert_response :success

    post "/login/login"
    assert_response :success
    assert_equal "Invalid login and/or password.", flash[:notice]

    post "/login/login", :login => user.login, :password => user.password
    assert_redirected_to "/home"

    follow_redirect!

    assert_tag :tag => "div", :attributes => { :id => "edit_form" }
  end
end

```

Note que o arquivo testa uma série de operações em vários *controllers*, seguindo redirecionamentos, postando dados, e verificando conteúdo quando necessário. Esse tipo de teste é bastante poderoso e pode ser, inclusive, aumentado para manipular diversas sessões simultâneas se esse for o caso.

Rodando o teste temos:

```

ronaldo@minerva:~/tmp/gtd$ rake test:integration
(in /home/ronaldo/tmp/gtd)
/usr/bin/ruby1.8 -Ilib:test "/usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader.rb"
"test/integration/login_test.rb"
Loaded suite /usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader
Started
.
Finished in 0.162952 seconds.

1 tests, 8 assertions, 0 failures, 0 errors
Loaded suite /usr/bin/rake
Started

Finished in 0.000314 seconds.

0 tests, 0 assertions, 0 failures, 0 errors

```

Isso encerra o nossa sessão sobre testes. Embora seja uma introdução bem rápida, esperamos que tenha sido o suficiente para mostrar como essa parte é importante e poderosa em suas aplicações. Lembre-se de começar a testar mesmo antes de desenvolver o que você precisa, e de criar testes suficientemente completos e complexos para verificar plenamente a sua aplicação testando desde métodos customizados em modelos de dados até interações entre vários *controllers*.

IMPLEMENTAÇÃO

Em algum momento, obviamente, você terá que implantar a sua aplicação em um servidor. No caso do Rails, existem dezenas de opções para implantação, considerando vários aspectos e necessidades do processo. O Rails pode rodar em contextos que vão de CGI e *cluster* de servidores Web customizados. Entrar em detalhes do como instalar o Rails sob um servidor como o Apache não é o propósito desse tutorial, mas podemos tecer algumas considerações sobre o processo de enviar a aplicação para o servidor.

Para realizar esse trabalho, existe uma ferramenta específica criada para o Rails que automatiza o processo, controlando versões e realizando outras tarefas específicas de implantação como, por exemplo, reiniciar o servidor Web em caso de necessidade. Essa ferramenta é chamada de *Capistrano*.

Para instalá-la, siga a rota comum:

```
ronaldo@minerva:~/tmp/gtd$ sudo gem install capistrano
```

Depois de instalado, você pode rodar o comando seguinte para ver se tudo a ferramenta está funcionando corretamente:

```
ronaldo@minerva:~/tmp/gtd$ cap -h
```

O que precisamos agora é adicionar a funcionalidade necessária à nossa aplicação com o comando abaixo:

```
ronaldo@minerva:~/tmp/gtd$ cap --apply-to . GTD
exists config
create config/deploy.rb
exists lib/tasks
create lib/tasks/capistrano.rake
Loaded suite /usr/bin/cap
Started
Finished in 0.00029 seconds.

0 tests, 0 assertions, 0 failures, 0 errors
```

Esse comando adiciona novas tarefas ao comando rake da aplicação, permitindo que configuremos e executemos novos comandos necessários para a implementação.

Vamos fazer alguns testes agora considerando somente uma aplicação simples, sem平衡amento de carga ou qualquer coisa como isso, embora o Capistrano seja poderoso o suficiente não só para enviar versões para um ou mais servidores, controlando-as corretamente.

Precisamos, primeiro, configurar nossas opções de implantação, editando o arquivo `deploy.rb`, que foi criado em nosso diretório `config`:

```

set :application, "gtd"
set :repository, "svn+ssh://yourdomain.com/repository/gtd/trunk"

role :web, "yourdomain.com"
role :app, "yourdomain.com"
role :db, "yourdomain.com", :primary => true

set :deploy_to, "/home/ronaldo/web/gtd/"
set :user, "ronaldo"

```

O arquivo acima foi reduzido ao mínimo, mostrando um repositório Subversio; servidores Web, de aplicação e de banco de dados configurado (no nosso caso o mesmo); o diretório para onde as versões serão enviadas; e o usuário remoto que executará esses comandos. A configuração acima obviamente assume que você possui um repositório Subversion do código que estamos rodando—sem o qual o comando não rodará.

Feito isso, precisamos agora rodar um comando para inicializar nossas estruturas:

```

ronaldo@minerva:~/tmp/gtd$ rake remote:exec ACTION=setup
(in /home/ronaldo/tmp/gtd)
  loading configuration /usr/lib/ruby/gems/1.8/gems/capistrano-
1.2.0/lib/capistrano/recipes/standard.rb
  loading configuration ./config/deploy.rb
* executing task setup
* executing "mkdir -p -m 775 /home/ronaldo/web/gtd/releases /home/ronaldo/web/gtd/shared/system
&&\n    mkdir -p -m 777 /home/ronaldo/web/gtd/shared/log &&\n        mkdir -p -m 777
/home/ronaldo/web/gtd/shared/pids"
  servers: ["yourdomain.com"]
Password: *****

[yourdomain.com] executing command
command finished

```

O comando acima cria o que é necessário para a implantação. Podemos agora enviar os nossos arquivos para o servidor.

```

ronaldo@minerva:~/tmp/gtd$ rake deploy
(in /home/ronaldo/tmp/gtd)
** Invoke deploy (first_time)
** Invoke remote:deploy (first_time)
** Execute remote:deploy
  loading configuration /usr/lib/ruby/gems/1.8/gems/capistrano-
1.2.0/lib/capistrano/recipes/standard.rb
  loading configuration ./config/deploy.rb
* executing task deploy
* executing task update
** transaction: start
* executing task update_code
* querying latest revision...
Password: *****
* executing "if [[ ! -d /home/ronaldo/web/gtd/releases/20061002064028 ]]; then\n          svn
co -q -r1 svn+ssh://yourdomain.com/repository/gtd/trunk /home/ronaldo/web/gtd/releases/20061002064028
&&\n          (test -e /home/ronaldo/web/gtd//revisions.log || (touch
/home/ronaldo/web/gtd//revisions.log && chmod 666 /home/ronaldo/web/gtd//revisions.log)) && echo
`date +"%Y-%m-%d %H:%M:%S\" ` $USER 1 20061002064028 >> /home/ronaldo/web/gtd//revisions.log;\nfi"
  servers: ["yourdomain.com"]
Password: *****

[yourdomain.com] executing command

```

```

** [out :: yourdomain.com] Password:
** [out :: yourdomain.com] subversion is asking for a password
** [out :: yourdomain.com] Password:
** [out :: yourdomain.com] subversion is asking for a password
  command finished
* executing "rm -rf /home/ronaldo/web/gtd/releases/20061002064028/log
/home/ronaldo/web/gtd/releases/20061002064028/public/system &&\n      ln -nfs
/home/ronaldo/web/gtd/shared/log /home/ronaldo/web/gtd/releases/20061002064028/log &&\n      ln -nfs
/home/ronaldo/web/gtd/shared/system /home/ronaldo/web/gtd/releases/20061002064028/public/system"
  servers: ["yourdomain.com"]
  [yourdomain.com] executing command
  command finished
* executing "test -d /home/ronaldo/web/gtd/shared/pids && \n      rm -rf
/home/ronaldo/web/gtd/releases/20061002064028/tmp/pids && \n      ln -nfs
/home/ronaldo/web/gtd/shared/pids /home/ronaldo/web/gtd/releases/20061002064028/tmp/pids; true"
  servers: ["yourdomain.com"]
  [yourdomain.com] executing command
  command finished
* executing task symlink
* executing "ls -x1 /home/ronaldo/web/gtd/releases"
  servers: ["yourdomain.com"]
  [yourdomain.com] executing command
  command finished
* executing "ln -nfs /home/ronaldo/web/gtd/releases/20061002064028 /home/ronaldo/web/gtd/current"
  servers: ["yourdomain.com"]
  [yourdomain.com] executing command
  command finished
** transaction: commit
* executing task restart
* executing "sudo /home/ronaldo/web/gtd/current/script/process/reaper"
  servers: ["yourdomain.com"]
  [yourdomain.com] executing command
** [out :: yourdomain.com] Couldn't find any process matching:
/home/ronaldo/web/gtd/current/public/dispatch.fcgi
  command finished
** Execute deploy

```

Se houver código no repositório, o resultado é que a versão mais recente é verificada e lançada no diretório correto da aplicação. Se houver processos ativos rodando com a versão antiga eles serão reiniciados conforme a necessidade. No caso acima, você pode ver que essa reinicialização falhou porque não configuramos um servidor Web.

Finalmente, se você olhar o diretório da aplicação, verá o seguinte agora:

```

root@bitbucket:/home/ronaldo/web/gtd# ls -l
total 12
lrwxrwxrwx  1 ronaldo ronaldo   45 2006-10-02 03:47 current ->
/home/ronaldo/web/gtd/releases/20061002064028
drwxrwxr-x  7 ronaldo ronaldo 4096 2006-10-02 03:47 releases
-rw-rw-rw-  1 ronaldo ronaldo  225 2006-10-02 03:47 revisions.log
drwxr-xr-x  5 ronaldo ronaldo 4096 2006-10-02 03:28 shared

```

Temos um diretório de versões e uma versão corrente.

Embora a ação acima não envie o banco de dados, uma das tarefas do Capistrano faz justamente isso. E existem dezenas de outras tarefas para controlar várias aspectos da implantação. É fácil ver como o uso do Capistrano facilita enormemente a implantação no servidor.

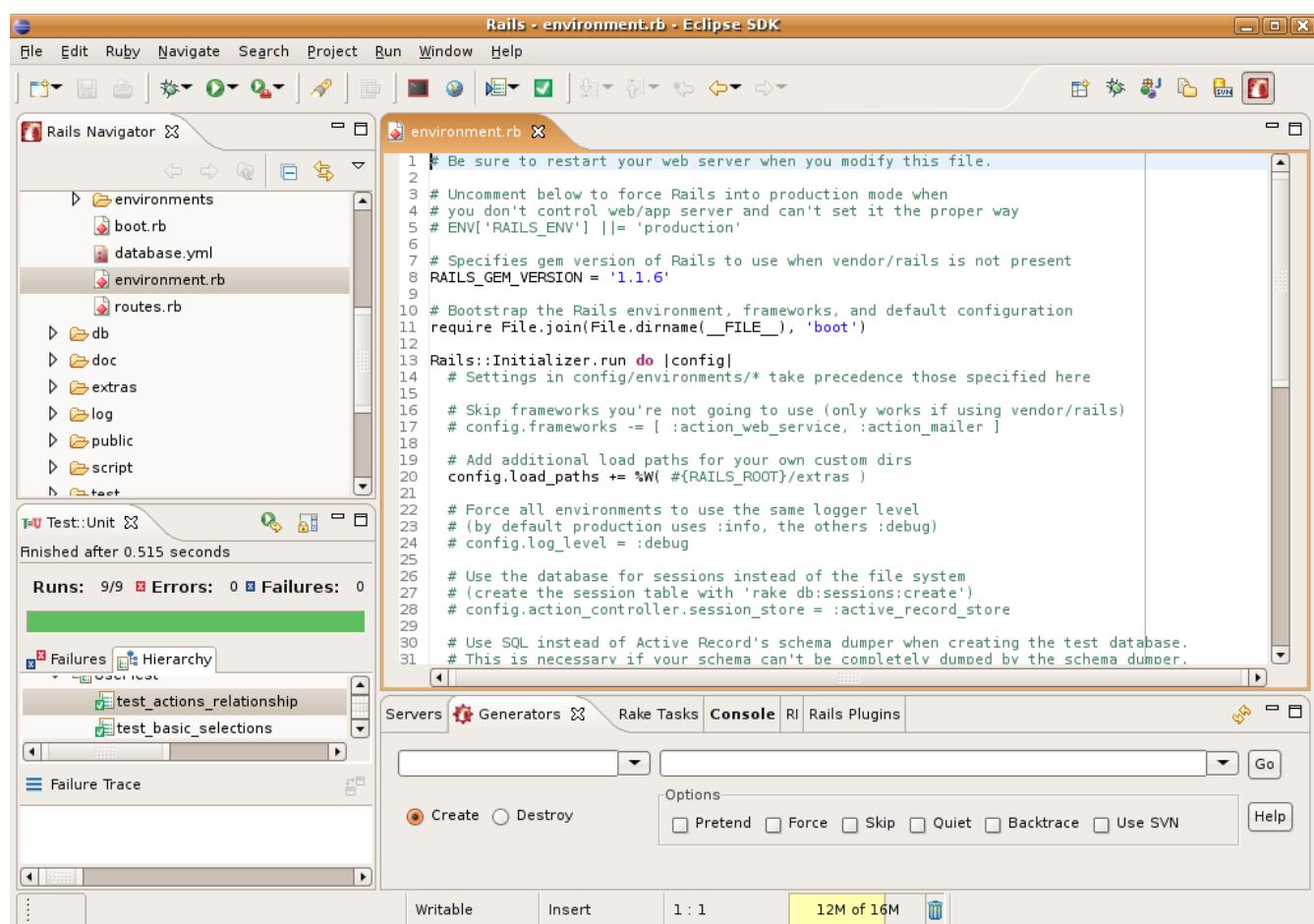
Para ler mais e entender todas as outras possibilidades da ferramenta, consulte a sua documentação no endereço abaixo:

<http://manuals.rubyonrails.com/read/book/17>

AMBIENTES DE DESENVOLVIMENTO

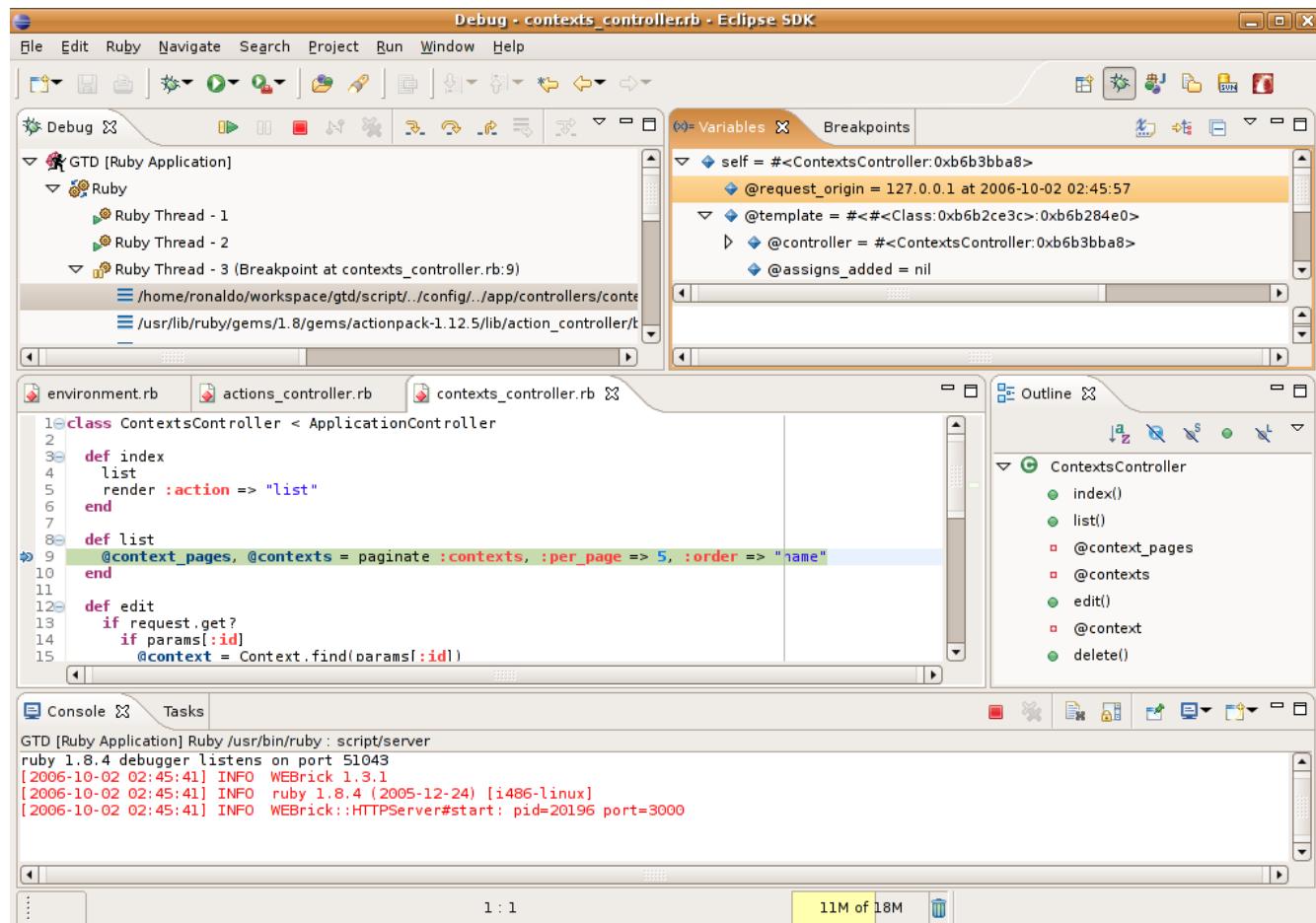
Desenvolver com fizemos até o momento, usando apenas a linha de comando e um editor de textos pode parecer conveniente mas ferramentas estão surgindo para ajudar a vida do desenvolvedor Rails, nos vários sistemas operacionais em que o *framework* está sendo usado.

Uma das ferramentas em especial, o RadRails, que é baseado no Eclipse, possui uma gama de características voltados para o Rails que poupa uma quantidade substancial de trabalho. Você pode encontrar o RadRails em <http://www.radrails.org/>, sendo que existem versões do mesmo compiladas previamente para Windows, Linux e Macintosh. Caso você queira, você pode integrar diretamente as funcionalidades do mesmo a uma instalação Eclipse comum, usando *update sites*. Essa última maneira é bem conveniente para aqueles que já estão usando o Eclipse para outros tipos de desenvolvimento.



A tela acima mostra visão tomada logo após rodarmos os testes de unidade da aplicação, que não tiveram nenhuma falha. Como podemos ver, temos navegação, edição com sintaxe própria, tarefas específicas do Rake, instalação de *plugins* integrada, documentação, e dezenas de outras pequenas funcionalidades que tornam o RadRails um excelente ambiente para projetos Rails.

Com um pouco de esforço, é possível também depurar aplicações Rails diretamente no RadRails, como mostra a tela abaixo:



Instruções para isso podem ser encontradas no próprio *site* do RadRails.

Eu estou usando o RadRails desde sua versão 0.2, lançada há quase um ano atrás—a versão atual é a 0.71—e estou bastante satisfeito com o ambiente. Apesar de alguns problemas em algumas versões intermediárias, a versão atual é uma ferramenta que vale a pena pelo menos testar.

Duas outras possibilidades que eu ainda não testei são, respectivamente:

RIDE-ME: um IDE Rails para Windows, familiar para usuários do Visual Studio.

<http://www.projectrideme.com/>

KDevelop: O IDE padrão para o KDE possui suporte ao Rails em suas versões mais recentes.

<http://www.kdevelop.org/>

RAKE A SEU SERVIÇO

Além de todas os comandos que vimos para o `rake` até o momento, outros comandos podem nos ser bem úteis. Um deles, por exemplo, são as estatísticas do projeto, que podemos ver abaixo:

```
ronaldo@minerva:~/tmp/gtd$ rake stats
(in /home/ronaldo/tmp/gtd)
+-----+-----+-----+-----+-----+-----+
| Name      | Lines | LOC | Classes | Methods | M/C | LOC/M |
+-----+-----+-----+-----+-----+-----+
| Helpers   | 23   | 21  | 0     | 1     | 0   | 19   |
| Controllers | 405  | 347 | 9     | 45    | 5   | 5    |
| Components | 0    | 0   | 0     | 0     | 0   | 0    |
| Functional tests | 155  | 114 | 16   | 25   | 1   | 2    |
| Models    | 105  | 72  | 7     | 4     | 0   | 16   |
| Unit tests | 128  | 89  | 7     | 12   | 1   | 5    |
| Libraries  | 0    | 0   | 0     | 0     | 0   | 0    |
| Integration tests | 29   | 18  | 1     | 1     | 1   | 16   |
+-----+-----+-----+-----+-----+-----+
| Total     | 845  | 661 | 40   | 88   | 2   | 5    |
+-----+-----+-----+-----+-----+-----+
Code LOC: 440      Test LOC: 221      Code to Test Ratio: 1:0.5
```

Não há nada de científico nesse resultado, mas ele serve com um bom lembrete—principalmente dos testes que precisamos implementar. Medir produtividade e código por linhas de código é algo considerado contraproducente hoje, mas comparar a quantidade de código existente por área da aplicação pode ser bem útil, especialmente depois de nos acostumarmos com os números que devemos esperar.

O comando abaixo nos mostra tudo o que o Rails pode fazer:

```
ronaldo@minerva:~/tmp/gtd$ rake --tasks
(in /home/ronaldo/tmp/gtd)
rake db:fixtures:load          # Load fixtures into the current environment's database. Load
specific fixtures using FIXTURES=x,y
rake db:migrate                # Migrate the database through scripts in db/migrate. Target specific
version with VERSION=x
rake db:schema:dump            # Create a db/schema.rb file that can be portably used against any DB
supported by AR
rake db:schema:load            # Load a schema.rb file into the database
rake db:sessions:clear          # Clear the sessions table
rake db:sessions:create         # Creates a sessions table for use with
CGI::Session::ActiveRecordStore
rake db:structure:dump          # Dump the database structure to a SQL file
rake db:test:clone              # Recreate the test database from the current environment's database
schema
...
rake test                      # Test all units and functionals
rake test:functionals           # Run tests for functionalsdb:test:prepare
rake test:integration           # Run tests for integrationdb:test:prepare
rake test:plugins                # Run tests for pluginsenvironment
rake test:recent                 # Run tests for recentdb:test:prepare
rake test:uncommitted             # Run tests for uncommiteddb:test:prepare
rake test:units                  # Run tests for unitsdb:test:prepare
```

...

Não estamos exibindo tudo, mas você pode investigar algumas tarefas que você usará com maior freqüência, além dos testes, é claro. Em especial, dê uma olhada nas seguintes tarefas: `db:structure:dump`, `rails:update`, e `doc:rails`.

VIVENDO NO LIMITE

O futuro do Rails parece estar garantido. O *framework* vem ganhando adeptos continuamente, livros estão sendo publicados sobre o mesmo em dezenas de idiomas e o progresso de novas versões é continuo. Algumas das novidades previstas para a versão 1.2, que está para sair a qualquer momento, incluem dois dos grandes tópicos em voga atualmente:

- Suporte padrão a REST, mencionado de passagem anteriormente;
- *ActiveResource*, uma nova adição à família de bibliotecas de suporte ao Rails que permite o acesso a objetos remotos de maneira transparente.

Caso você queira experimentar com algumas dessas características e também ver o que há de novo nas outras áreas do Rails, você pode atualizar a versão do Rails em sua aplicação para a versão atualmente presente no repositório com um simples comando:

```
ronaldo@minerva:~/tmp$ rake rails:freeze:edge
```

Esse comando baixa a revisão atual do Rails, colocando-a no diretório `vendor/rails`. Obviamente, sendo o código mais recente, há sempre a possibilidade de que ele contenha *bugs* que impediram que sua aplicação rode. Portanto, use esse código somente para testes e nunca para produção.

Para reverter o comando acima, use:

```
rake rails:unfreeze
```

O QUE FALTA

Nesse tutorial não foram incluídos alguns assuntos porque consideramos que não conseguiríamos tratá-los de maneira adequada, geralmente por dependerem de um conhecimento externo que não se enquadra no propósito desse documento.

Entre esses assuntos, o mais importante seja talvez o uso de *web services* em Rails—interessante em si próprio, mas cujo uso é relativamente específico e cuja dependência de vários padrões tornou inviável um tratamento rápido do mesmo. Da mesma forma, não tratamos de *engines* e desenvolvimentos similares, que são extensões ao Rails.

Como esse tutorial também não tem o objetivo de ser um material de referência nem uma obra exaustiva sobre o Rails, somente as funções estritamente necessárias para o código foram mencionadas. O Rails é um *framework* de tamanho considerável e não estaria no espírito desse tutorial documentar cada classe, método e variável presentes no mesmo. Para isso, existem livros já publicados sobre o assunto que você pode encontrar facilmente em livrarias técnicas.

Espero que o resto do tutorial seja o suficiente para compensar essas faltas.

Conclusão

Terminamos aqui o o tutorial.

Todos os erros presentes no texto e no código são, obviamente, meus. Eu tomei um cuidado especial nas seções que envolvem programação, procurando sempre mostrar entrada e saída exatas do que estava fazendo, e voltando para corrigir minunciosamente o que eu mudara, mas sempre há a possibilidade de que alguma coisa tenha passado. Se você descobrir algum problema, entre em contato no endereço deixado abaixo.

De acordo com a licença escolhida para este documento—que você pode ler no apêndice A—você é livre para copiar, distribuir e/ou modificar esse documento da maneira que achar melhor. Você pode, inclusive, publicar um livro usando o material completo, com ou sem modificações, no seu próprio nome, e vendê-lo pelo preço que achar justo. Vá em frente e faça o que bem entender: a licença lhe dá plena liberdade para isso.

Caso queira me enviar um e-mail com sugestões, críticas ou quaisquer outros comentários, não hesite. O endereço para contato é: ronaldo@reflectivesurface.com. Com exceção de pedidos de ajuda que não tem a ver com o tutorial em si—que eu teria o maior prazer em atender se tivesse tempo, mas, infelizmente, esse não é caso—eu farei o melhor para responder todos e-mails. A minha empresa também está disponível para trabalhos em Ruby on Rails caso haja interesse.

No mais, compartilhe e divirta-se.

Apêndice A: GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

O. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a worldwide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the

Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the

Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your

Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a

particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

HOW TO USE THIS LICENSE FOR YOUR DOCUMENTS

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.