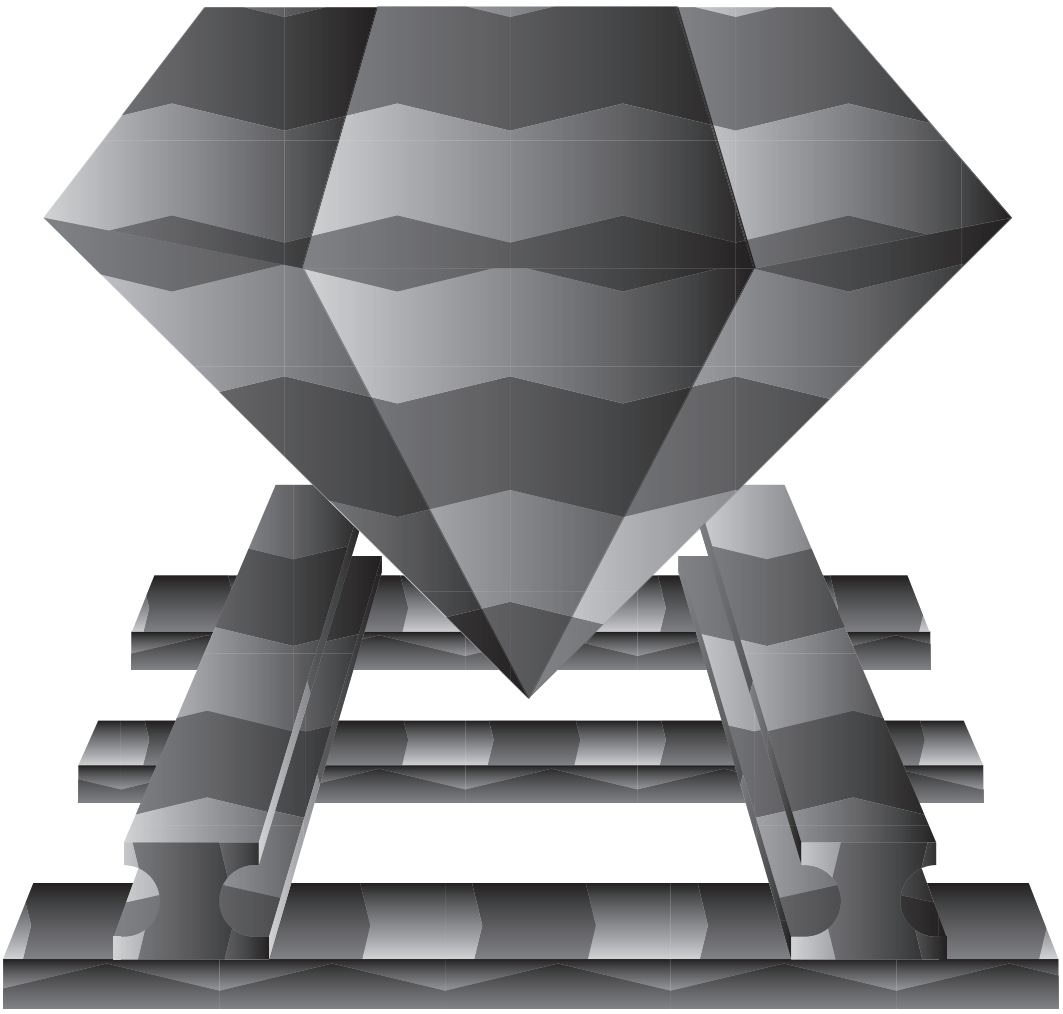
RR-71

Desenvolvimento Ágil para Web 2.0 com Ruby on Rails



Caelum
Ensino e Inovação
www.caelum.com.br

Caelum
Ensino e Inovação

Conheça mais da Caelum.

Cursos Online

www.caelum.com.br/online

Casa do Código

Livros para o programador

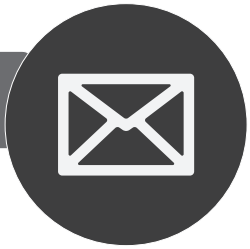
www.casadocodigo.com.br

Blog Caelum

blog.caelum.com.br

Newsletter

www.caelum.com.br/newsletter



Facebook

www.facebook.com/caelumbr

Twitter

twitter.com/caelum

Conheça alguns de nossos cursos

FJ-11:
Java e Orientação a objetos

FJ-25:
Persistência com JPA2 e Hibernate

FJ-16:
Laboratório Java com Testes, XML e Design Patterns

FJ-26:
Laboratório Web com JSF2 e CDI

FJ-19:
Preparatório para Certificação de Programador Java

FJ-31:
Java EE avançado e Web Services

FJ-21:
Java para Desenvolvimento Web

FJ-91:
Arquitetura e Design de Projetos Java

RR-71:
Desenvolvimento Ágil para Web 2.0 com Ruby on Rails

RR-75:
Ruby e Rails avançados: lidando com problemas do dia a dia

Para mais informações e outros cursos, visite: caelum.com.br/cursos

- ✓ Mais de 8000 alunos treinados;
- ✓ Reconhecida nacionalmente;
- ✓ Conteúdos atualizados para o mercado e para sua carreira;
- ✓ Aulas com metodologia e didática cuidadosamente preparadas;
- ✓ Ativa participação nas comunidades Java, Rails e Scrum;
- ✓ Salas de aula bem equipadas;
- ✓ Instrutores qualificados e experientes;
- ✓ Apostilas disponíveis no site.

Caelum
Ensino e Inovação

Sobre esta apostila

Esta apostila da Caelum visa ensinar de uma maneira elegante, mostrando apenas o que é necessário e quando é necessário, no momento certo, poupando o leitor de assuntos que não costumam ser de seu interesse em determinadas fases do aprendizado.

A Caelum espera que você aproveite esse material. Todos os comentários, críticas e sugestões serão muito bem-vindos.

Essa apostila é constantemente atualizada e disponibilizada no site da Caelum. Sempre consulte o site para novas versões e, ao invés de anexar o PDF para enviar a um amigo, indique o site para que ele possa sempre baixar as últimas versões. Você pode conferir o código de versão da apostila logo no final do índice.

Baixe sempre a versão mais nova em: www.caelum.com.br/apostilas

www.caelum.com.br

Sumário

1 Agilidade na Web **1**

1.1 A agilidade . . . 1
1.2 A comunidade Rails . . . 2
1.3 Bibliografia . . . 2
1.4 Tirando dúvidas . . . 3
1.5 Para onde ir depois? . . . 3

2 A linguagem Ruby **4**

2.1 A história do Ruby e suas características . . . 4
2.2 Instalação do interpretador . . . 4
2.3 RubyGems . . . 5
2.4 Bundler . . . 6
2.5 Outras implementações de interpretadores Ruby . . . 7

3 Ruby básico **10**

3.1 Aprender Ruby? . . . 10
3.2 Executando código Ruby no Terminal: IRB e arquivos .rb . . . 10
3.3 Variáveis, Strings e Comentários . . . 11
3.4 Variáveis e atribuições . . . 12
3.5 Tipagem . . . 13
3.6 Exercícios - Variáveis e Atribuições . . . 13
3.7 String . . . 14
3.8 Exercícios - Strings . . . 16
3.9 Tipos e operações básicas . . . 17
3.10 Exercícios - Tipos . . . 19
3.11 Estruturas de controle . . . 20
3.12 Exercícios - Estruturas de controle e Regexp . . . 22
3.13 Desafios . . . 23

4 Mais Ruby: classes, objetos e métodos **24**

4.1 Mundo orientado a objetos . . . 24
4.2 Métodos comuns . . . 25
4.3 Definição de métodos . . . 25
4.4 Exercícios - Métodos . . . 26
4.5 Discussão: Enviando mensagens aos objetos . . . 27
4.6 Classes . . . 27
4.7 Exercícios - Classes . . . 28
4.8 Desafio: Classes abertas . . . 29
4.9 self . . . 29

4.10 Desafio: self e o método puts . . . 29
4.11 Atributos e propriedades: acessores e modificadores . . . 30
4.12 Syntax Sugar: facilitando a sintaxe . . . 31
4.13 Exercícios - Atributos e propriedades . . . 32
4.14 Coleções . . . 34
4.15 Exemplo: múltiplos parâmetros . . . 35
4.16 Hashes . . . 36
4.17 Exercícios - Arrays e Hashes . . . 38
4.18 Blocos e Programação Funcional . . . 40
4.19 Exercícios - Blocos . . . 45
4.20 Para saber mais: Mais sobre blocos . . . 45
4.21 Desafio: Usando blocos . . . 46
4.22 Manipulando erros e exceptions . . . 46
4.23 Exercício: Manipulando exceptions . . . 47
4.24 Arquivos com código fonte ruby . . . 48
4.25 Para saber mais: um pouco de IO . . . 50

5 Metaprogramação **51**
5.1 Métodos de Classe . . . 51
5.2 Para saber mais: Singleton Classes . . . 52
5.3 Exercícios - Ruby Object Model . . . 53
5.4 Convenções . . . 54
5.5 Polimorfismo . . . 55
5.6 Exercícios - Duck Typing . . . 56
5.7 Modulos . . . 58
5.8 Metaprogramação . . . 59
5.9 Exercícios - Metaprogramação . . . 61

6 Ruby on Rails **63**
6.1 Ruby On Rails . . . 63
6.2 Ambiente de Desenvolvimento . . . 65
6.3 Criando um novo projeto Rails . . . 66
6.4 Exercícios - Iniciando o projeto . . . 67
6.5 Estrutura dos diretórios . . . 69
6.6 O Banco de dados . . . 70
6.7 Exercícios - Criando o banco de dados . . . 70
6.8 A base da construção: Scaffold . . . 72
6.9 Exercícios - Scaffold . . . 72
6.10 Gerar as tabelas . . . 72
6.11 Exercícios - Migrar tabela . . . 73
6.12 Server . . . 74
6.13 Exercícios - Iniciando o servidor . . . 75
6.14 Documentação do Rails . . . 75

6.15 Exercício opcional - Utilizando a documentação . . . 77

7 Active Record **78**
7.1 Motivação . . . 78
7.2 Exercícios: Controle de Restaurantes . . . 78
7.3 Modelo - O "M" do MVC . . . 79
7.4 ActiveRecord . . . 80
7.5 Rake . . . 80
7.6 Criando Modelos . . . 81
7.7 Migrations . . . 82
7.8 Exercícios: Criando os modelos . . . 83
7.9 Manipulando nossos modelos pelo console . . . 85
7.10 Exercícios: Manipulando registros . . . 88
7.11 Exercícios Opcionais . . . 90
7.12 Finders . . . 90
7.13 Exercícios: Buscas dinâmicas . . . 91
7.14 Validações . . . 92
7.15 Exercícios: Validações . . . 94
7.16 Pluralização com inflections . . . 95
7.17 Exercícios - Completando nosso modelo . . . 96
7.18 Exercícios - Criando o Modelo de Qualificação . . . 100
7.19 Relacionamentos . . . 103
7.20 Para Saber Mais: Auto-relacionamento . . . 105
7.21 Para Saber Mais: Cache . . . 105
7.22 Exercícios - Relacionamentos . . . 105
7.23 Para Saber Mais - Eager Loading . . . 109
7.24 Para Saber Mais - Named Scopes . . . 110
7.25 Para Saber Mais - Modules . . . 111

8 Rotas **112**
8.1 routes.rb . . . 112
8.2 Pretty URLs . . . 113
8.3 Named Routes . . . 114
8.4 REST - resources . . . 114
8.5 Actions extras em Resources . . . 116
8.6 Para Saber Mais - Nested Resources . . . 117

9 Controllers e Views **118**
9.1 O "V" e o "C" do MVC . . . 118
9.2 Hello World . . . 118
9.3 Exercícios: Criando o controlador . . . 119
9.4 Trabalhando com a View: O ERB . . . 120
9.5 Entendendo melhor o CRUD . . . 123

9.6 A action index . . . 123
9.7 Exercícios: Listagem de restaurantes . . . 124
9.8 Helper . . . 125
9.9 A action show . . . 126
9.10 Exercícios: Visualizando um restaurante . . . 127
9.11 A action destroy . . . 128
9.12 Redirecionamento de Action . . . 128
9.13 Exercícios: Deletando um restaurante . . . 129
9.14 Helpers para formulários . . . 130
9.15 A action new . . . 132
9.16 Exercícios: Página para criação de um novo restaurante . . . 132
9.17 Recebendo um parâmetro por um formulário . . . 132
9.18 A action create . . . 134
9.19 Exercícios: Persistindo um restaurante . . . 134
9.20 A action edit . . . 134
9.21 A action update . . . 135
9.22 Atualizando um restaurante . . . 135
9.23 Exercícios Opcionais: linkando melhor as views . . . 136
9.24 Mostrando erros ao usuário . . . 137
9.25 Exercícios: Tratamento de erros na criação . . . 138
9.26 Mensagens de erro na atualização . . . 139
9.27 Exercícios: Tratamento de erros na atualização . . . 140
9.28 Partial . . . 141
9.29 Exercícios: Reaproveitando fragmentos de ERB . . . 141
9.30 Respondendo em outros formatos como XML ou JSON . . . 142
9.31 Para saber mais: Outros handlers . . . 143
9.32 Exercícios: Disponibilizando restaurantes como XML e JSON . . . 144
9.33 Exercícios Opcionais: Outras actions respondendo XML e JSON . . . 145
9.34 Filtros . . . 145

10 Completando o Sistema **148**
10.1 Um pouco mais sobre o Scaffold . . . 148
10.2 Exercícios: Completando nosso domínio . . . 148
10.3 Selecionando Clientes e Restaurante no form de Qualificações . . . 151
10.4 Exercícios: Formulário com collection_select . . . 152
10.5 Exercícios Opcionais: Refatorando para respeitarmos o MVC . . . 152
10.6 Exercícios Opcionais: Exibindo nomes ao invés de números . . . 155
10.7 Mais sobre os controllers . . . 159

11 Calculations **162**
11.1 Métodos . . . 162
11.2 Média . . . 162
11.3 Exercícios . . . 163

12 Associações Polimórficas **165**
12.1 Nosso problema . . . 165
12.2 Alterando o banco de dados . . . 166
12.3 Exercícios: Criando modelo de comentário . . . 167

13 Mais sobre views **172**
13.1 Helpers customizados . . . 172
13.2 Exercícios: Formatando valores monetários . . . 172
13.3 Partials e a opção locals . . . 173
13.4 Exercícios: Formulário para criar um novo comentário . . . 175
13.5 Partials de coleções . . . 176
13.6 Exercícios: Listando os comentários de um comentável . . . 177
13.7 Layouts . . . 178
13.8 Exercícios: Criando um menu de navegação principal . . . 178
13.9 Exercícios Opcionais: Menu como partial . . . 179
13.10 Aplicar CSS em apenas algumas views . . . 180
13.11 Exercícios: Aplicando CSS para os comentários . . . 181

14 Ajax com Rails **182**
14.1 Utilizando AJAX para remoção de comentários . . . 182
14.2 Exercícios: Link de remover comentário usando AJAX . . . 185
14.3 Adicionando comentários dinamicamente . . . 187
14.4 Exercícios: AJAX no formulário de comentários . . . 189

15 Algumas Gems Importantes **191**
15.1 Engines . . . 191
15.2 Exercícios: Paginação com Kaminari . . . 192
15.3 File Uploads: Paperclip . . . 193
15.4 Exercícios: Possibilitando upload de imagens . . . 194
15.5 Nokogiri . . . 195
15.6 Exercícios: Extraindo dados do Twitter . . . 195

16 Apêndice: Testes **197**
16.1 O Porquê dos testes? . . . 197
16.2 `Test::Unit` . . . 197
16.3 Exercícios - Teste do modelo . . . 200
16.4 Exercícios - Teste do controller . . . 201
16.5 RSpec . . . 202
16.6 Cucumber, o novo Story Runner . . . 206

17 Apêndice: Rotas e Rack **210**
17.1 Rack . . . 210
17.2 Exercícios - Testando o Rack . . . 211

17.3 Rails e o Rack . . . 211
17.4 Exercícios - Criando um rota para uma aplicação Rack . . . 213

18 Apêndice: Design Patterns em Ruby 214
18.1 Singleton . . . 214
18.2 Exercícios: Singleton . . . 215
18.3 Template Method . . . 216
18.4 Exercícios: Template Method . . . 217
18.5 Design Patterns: Observer . . . 218
18.6 Desafio: Design Pattern - Observer . . . 220

19 Apêndice: Integrando Java e Ruby 221
19.1 O Projeto . . . 221
19.2 Testando o JRuby . . . 221
19.3 Exercícios . . . 222
19.4 Compilando ruby para .class com Jruby . . . 222
19.5 Rodando o .class do ruby na JVM . . . 223
19.6 Importando um bytecode(.class) criado pelo jruby . . . 223
19.7 Importando classes do Java para sua aplicação JRuby . . . 223
19.8 Testando o JRuby com Swing . . . 225

20 Apêndice: Deployment 227
20.1 Webrick . . . 227
20.2 CGI . . . 228
20.3 FCGI - FastCGI . . . 228
20.4 Lighttpd e Litespeed . . . 228
20.5 Mongrel . . . 229
20.6 Proxies Reversos . . . 229
20.7 Phusion Passenger (mod_rails) . . . 230
20.8 Ruby Enterprise Edition . . . 230
20.9 Exercícios: Deploy com Apache e Passenger . . . 231

Índice Remissivo 232

Versão: 16.0.6

Capítulo 1

Agilidade na Web

"Não são os milagres que inclinam o realista para a fé. O verdadeiro realista, caso não creia, sempre encontrará em si força e capacidade para não acreditar no milagre, e se o milagre se apresenta diante dele como fato irrefutável, é mais fácil ele descrer de seus sentidos que admitir o fato"
– Fiodór Dostoiévski, em Irmãos Karamazov

1.1 A agilidade

Quais são os problemas mais frequentes no desenvolvimento web? Seriam os problemas com AJAX? Escrever SQL? Tempo demais para gerar os CRUDs básicos?

Com tudo isso em mente, David Heinemeier Hansson, trabalhando na *37Signals*, começou a procurar uma linguagem de programação que pudesse utilizar para desenvolver os projetos de sua empresa. Mais ainda, criou um framework web para essa linguagem, que permitiria a ele escrever uma aplicação web de maneira simples e elegante.

Em agosto de 2010 foi lançada a versão final do Rails 3 e a Caelum ministra turmas com o material atualizado para esta versão desde então.

O que possibilita toda essa simplicidade do Rails são os recursos poderosos que Ruby oferece e que deram toda a simplicidade ao Rails. Esses recursos proporcionados pela linguagem Ruby são fundamentais de serem compreendidos por todos que desejam se tornar bons desenvolvedores Rails e por isso o começo desse curso foca bastante em apresentar as características da linguagem e seus diferenciais.

Um exemplo clássico da importância de conhecer mais a fundo a linguagem Ruby está em desvendar a "magia negra" por trás do Rails. Conceitos como metaprogramação, onde código é criado dinamicamente, são essenciais para o entendimento de qualquer sistema desenvolvido em Rails. É a meta programação que permite, por exemplo, que tenhamos classes extremamente enxutas e que garante o relacionamento entre as

tabelas do banco de dados com nossas classes de modelo sem a necessidade de nenhuma linha de código, apenas usando de convenções.

Esse curso apresenta ainda os conceitos de programação funcional, uso de blocos, duck typing, enfim, tudo o que é necessário para a formação da base de conceitos que serão utilizados ao longo do curso e da vida como um desenvolvedor Rails.

1.2 A comunidade Rails

A comunidade Rails é hoje uma das mais ativas e unidas do Brasil. Cerca de 10 eventos acontecem anualmente com o único propósito de difundir conhecimento e unir os desenvolvedores. Um exemplo dessa força é o Ruby Conf, maior evento de Ruby da America Latina, com presença dos maiores nomes nacionais e internacionais de Ruby on Rails, e a presença de uma track dedicada ao Ruby na QCon São Paulo.

Além dos eventos, diversos blogs sobre Rails tem ajudado diversos programadores a desvendar esse novo universo:

- http://blog.caelum.com.br/ - Blog da Caelum
- http://andersonleite.com.br/ - Anderson Leite
- http://yehudakatz.com/ - Yehuda Katz
- http://fabiokung.com/ - Fabio Kung
- http://akitaonrails.com/ - Fábio Akita
- http://blog.plataformatec.com.br/ - José Valim
- http://nomedojogo.com/ - Carlos Brando
- http://devblog.avdi.org/ - Avdi Grimm
- http://blog.hasmanythrough.com/ - Josh Susser
- http://rubyflow.com/ - Agrega conteúdo de vários sites

A Caelum aposta no Rails desde 2007, quando criamos o primeiro curso a respeito. E o ano de 2009 marcou o Ruby on Rails no Brasil, ano em que ele foi adotado por diversas empresas grandes e até mesmo órgãos do governo, como mencionado num post em nosso blog no começo do mesmo ano:

http://blog.caelum.com.br/2009/01/19/2009-ano-do-ruby-on-rails-no-brasil/

1.3 Bibliografia

- **Ruby on Rails: Coloque sua aplicação web nos trilhos - Vinícius Baggio Fuentes**Criado por um dos organizadores do GURU-SP, o livro mostra com exemplos práticos e explicações didáticas, como criar

uma aplicação Rails do começo ao fim. Mostra recursos importantes como o envio de e-mails, explica o MVC e ainda passa por diversos itens úteis do dia-dia, com deploy no Cloud. Você encontra o livro, inclusive em versão e-book em http://www.casadocodigo.com.br

- **Agile Web Development with Rails - Sam Ruby, Dave Thomas, David Heinemeier Hansson**Esse é o livro referência no aprendizado de Ruby on Rails, criado pelo autor do framework. Aqui, ele mostra através de um projeto, os principais conceitos e passos no desenvolvimento de uma aplicação completa. Existe uma versão em andamento para Rails 3.
- **Programming Ruby: The Pragmatic Programmers' Guide - Dave Thomas, Chad Fowler, Andy Hunt**Conhecido como "Pickaxe", esse livro pode ser considerado a bíblia do programador Ruby. Cobre toda a especificação da linguagem e procura desvendar toda a "magia" do Ruby.
- **The Pragmatic Programmer: From Journeyman to Master - Andrew Hunt, David Thomas** As melhores práticas para ser um bom desenvolvedor: desde o uso de versionamento, ao bom uso do logging, debug, nomenclaturas, como consertar bugs, etc.Existe ainda um post no blog da Caelum sobre livros que todo desenvolvedor Rails deve ler: http://blog.caelum.com.br/2009/08/25/a-trinca-de-ases-do-programador-rails/

1.4 Tirando dúvidas

Para tirar dúvidas dos exercícios, ou de Ruby e Rails em geral, recomendamos se inscrever na lista do GURU-SP (http://groups.google.com/group/ruby-sp), onde sua dúvida será respondida prontamente.

Também recomendamos duas outras listas:

- http://groups.google.com/group/rubyonrails-talk
- http://groups.google.com/group/rails-br

Fora isso, sinta-se à vontade para entrar em contato com seu instrutor e tirar todas as dúvidas que tiver durante o curso.

O fórum do GUJ.com.br, conceituado em java, possui também um subfórum de Rails:

http://www.guj.com.br/

1.5 Para onde ir depois?

Além de fazer nossos cursos de Rails, você deve participar ativamente da comunidade. Ela é muito viva e ativa, e as novidades aparecem rapidamente. Se você ainda não tinha hábito de participar de fóruns, listas e blogs, essa é uma grande oportunidade.

Há ainda a possibilidade de participar de projetos opensource, e de você criar gems e plugins pro Rails que sejam úteis a toda comunidade.

Capítulo 2

A linguagem Ruby

"Rails is the killer app for Ruby."
– Yukihiro Matsumoto, Criador da linguagem Ruby

Nos primeiros capítulos conheceremos a poderosa linguagem de programação Ruby, base para completo entendimento do framework Ruby on Rails. Uma linguagem dinâmica e poderosa como o Ruby vai se mostrar útil não apenas para o uso no Rails, como também para outras tarefas e scripts do dia a dia.

Veremos a história, o contexto, suas versões e interpretadores e uma iniciação a sintaxe da linguagem.

2.1 A história do Ruby e suas características

Ruby foi apresentada ao público pela primeira vez em 1995, pelo seu criador: **Yukihiro Matsumoto**, mundialmente conhecido como **Matz**. É uma linguagem orientada a objetos, com tipagem forte e dinâmica. Curiosamente é uma das únicas linguagens nascidas fora do eixo EUA - Europa que atingiram enorme sucesso comercial.

Uma de suas principais características é a expressividade que possui. Teve-se como objetivo desde o início que fosse uma linguagem muito simples de ler e ser entendida, para facilitar o desenvolvimento e manutenção de sistemas escritos com ela.

Ruby é uma linguagem interpretada e, como tal, necessita da instalação de um interpretador em sua máquina antes de executar algum programa.

2.2 Instalação do interpretador

Antes da linguagem Ruby se tornar popular, existia apenas um interpretador disponível: o escrito pelo próprio Matz, em C. É um interpretador simples, sem nenhum gerenciamento de memória muito complexo,

nem características modernas de interpretadores como a compilação em tempo de execução (conhecida como JIT). Hoje a versão mais difundida é a 1.9, também conhecida como **YARV** (Yet Another Ruby VM), já baseada em uma máquina virtual com recursos mais avançados.

A maioria das distribuições Linux possuem o pacote de uma das última versões estáveis pronto para ser instalado. O exemplo mais comum é o de instalação para o Ubuntu:

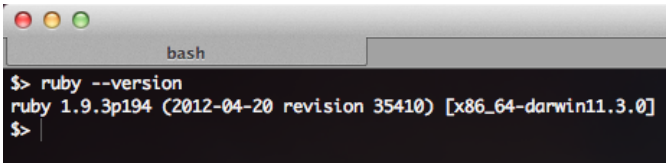
```
sudo apt-get install ruby1.9.1 ruby1.9.1-dev
```

Apesar de existirem soluções prontas para a instalação do Ruby em diversas plataformas (*one-click-installers* e até mesmo gerenciadores de versões de Ruby), sempre é possível baixá-lo pelo site oficial:

http://ruby-lang.org

Após a instalação, não deixe de conferir se o interpretador está disponível na sua variável de ambiente *PATH*:

```
$ ruby --version
ruby 1.9.3p194 (2012-04-20 revision 35410) [x86_64-darwin11.3.0]
```



2.3 RubyGems

O Ruby possui um gerenciador de pacotes bastante avançado, flexível e eficiente: **RubyGems**. As *gems* podem ser vistas como bibliotecas reutilizáveis de código Ruby, que podem até conter algum código nativo (em C, Java, .Net). São análogos aos *jars* no ambiente Java, ou os *assemblies* do mundo .Net. RubyGems é um sistema gerenciador de pacotes comparável a qualquer um do mundo *NIX, como os `.debs` do `apt-get`, os `rpms` do `yum`, entre outros.

Para sermos capazes de instalar e utilizar as centenas de *gems* disponíveis, precisamos instalar além do interpretador Ruby, o RubyGems. Como exemplo simples, temos o comando para instalação no Ubuntu:

```
sudo apt-get install rubygems1.9.1
```

> **Mais sobre instalação**
>
> No final da apostila, apresentamos um apêndice exclusivo sobre a instalação completa do interpretador e ferramentas essenciais nas plataformas Linux, Mac e Windows.

Após a instalação do RubyGems, podemos instalar os pacotes necessários para a criação de uma aplicação Rails com o comando **rails new**:

```
rails new nome_da_aplicacao
```

2.4 Bundler

Ao desenvolver novas aplicações utilizando Ruby, notaremos que uma série de funcionalidades serão necessárias - ler e parsear JSON, fazer autenticação de usuário, entre outras coisas. A maioria dessas funcionalidades já foi implementada em alguma *gem*, e para usufruir desses recursos basta colocar a gem em nossa aplicação.

Nossos projetos podem precisar de uma ou mais gems para funcionar corretamente. Se quiséssemos compartilhar nosso projeto com a comunidade, como faríamos para fixar as gems necessárias para o funcionamento do projeto? Escreveríamos um arquivo de texto especificando quais gems o outro desenvolvedor deve instalar? Esse seria um processo manual? Há diversas formas de fazermos um bom controle de versão de código, mas como poderíamos, no mundo Ruby, controlar as *dependências* de gems necessárias ao nosso projeto?

Uma forma eficaz de controlar as dependências de nosso projeto Ruby é utilizar uma gem chamada *Bundler*. Instalar o *bundler* é bem simples. Basta rodar o comando:

```
gem install bundler
```

Com o Bundler, declaramos as dependências necessárias em um arquivo chamado *Gemfile*. Esse aquivo deverá conter, primeiramente, a fonte de onde onde o Bundler deve obter as gems e em seguida a declaração das dependências que usaremos no projeto.

```
source "http://rubygems.org"

gem "rails"
gem "devise"
```

Note que a declaração *source* indica de onde o Bundler obterá as gems (no caso mais comum, de http://rubygems.org) e a declaração *gem* indica a gem declarada como dependência.

Podemos então rodar o comando `bundle` (atalho para o comando `bundle install`) para obter as gems necessárias para o nosso projeto. Note que ao estruturar a declaração das gems dessa forma não declaramos versões específicas para as gems, portanto a versão mais recente de cada uma dela será baixada para nosso repositório de gems local. Poderíamos fazer o seguinte:

```
gem "rails", "3.1.0"
```

Dessa forma explicitaríamos o uso da versão 3.1.0 da gem `rails`. Uma outra forma válida seria:

```
gem "rails", "~> 3.1.0"
```

Assim obteríamos a gem em uma versão maior ou igual à versão 3.1.0, mas menor que a versão 3.2.0.

Ao rodar o comando `bundle` será gerado um novo arquivo chamado *Gemfile.lock*, que especifica todas as gems obtidas para aquele Gemfile e sua respectiva versão baixada. O Gemfile.lock é uma boa alternativa para congelar as versões das gems a serem utilizadas, uma vez que ao rodarmos o comando `bundle` sobre a presença de um Gemfile.lock, as versões presentes nesse arquivo serão utilizadas para especificar as gems a serem baixadas.

2.5 Outras implementações de interpretadores Ruby

Com a popularização da linguagem Ruby, principalmente após o surgimento do Ruby on Rails, implementações alternativas da linguagem começaram a surgir. A maioria delas segue uma tendência natural de serem baseados em uma Máquina Virtual ao invés de serem interpretadores simples. Algumas implementações possuem até compiladores completos, que transformam o código Ruby em alguma linguagem intermediária a ser interpretada por uma máquina virtual.

A principal vantagem das máquinas virtuais é facilitar o suporte em diferentes plataformas. Além disso, ter código intermediário permite otimização do código em tempo de execução, feito através da **JIT**.

JRuby

JRuby foi a primeira implementação alternativa completa da versão 1.8.6 do Ruby e é a principal implementação da linguagem Java para a JVM. Com o tempo ganhou compatibilidade com as versões 1.8.7 e 1.9.2 na mesma implementação.

Como roda na JVM, não é um simples interpretador, já que também opera nos modos de compilação AOT (*Ahead Of Time*) e JIT (*Just In Time*), além do modo interpretador tradicional *Tree Walker*.

Uma de suas principais vantagens é a interoperabilidade com código Java existente, além de aproveitar todas as vantagens de uma das plataformas de execução de código mais maduras (Garbage Collector, Threads nativas, entre outras).

Muitas empresas apoiam o projeto, além da própria Oracle (após ter adquirido a Sun Microsystems), outras empresas de expressão como IBM, Borland e ThoughtWorks mantém alguns projetos na plataforma e algumas delas têm funcionários dedicados ao projeto.

http://jruby.org

IronRuby

A comunidade .Net também não ignora o sucesso da linguagem e patrocina o projeto **IronRuby**, mantido pela própria *Microsoft*. IronRuby foi um dos primeiros projetos verdadeiramente de código aberto dentro da

Microsoft.

http://ironruby.net

Rubinius

Criada por Evan Phoenix, **Rubinius** é um dos projetos que tem recebido mais atenção da comunidade Ruby, por ter o objetivo de criar a implementação de Ruby com a maior parte possível do código em Ruby. Além disso, trouxe ideias de máquinas virtuais do SmallTalk, possuindo um conjunto de instruções (bytecode) próprio e implementada em C/C++.

http://rubini.us

RubySpec

O projeto Rubinius possui uma quantidade de testes enorme, escritos em Ruby, o que incentivou a iniciativa de especificar a linguagem Ruby. O projeto RubySpec (http://rubyspec.org/) é um acordo entre os vários implementadores da linguagem Ruby para especificar as características da linguagem Ruby e seu comportamento, através de código executável, que funciona como um **TCK** (*Test Compatibility Kit*).

RubySpec tem origem na suíte de testes de unidade do projeto Rubinius, escritos com uma versão mínima do RSpec, conhecida como **MSpec**. O RSpec é uma ferramenta para descrição de especificações no estilo pregado pelo *Behavior Driven Development*.

Avi Bryant durante seu *keynote* na **RailsConf de 2007** lançou um desafio à comunidade:

*"I'm from the future, I know how this story ends. All the people who are saying you can't implement Ruby on a fast virtual machine are wrong. That machine already exists today, it's called **Gemstone**, and it could certainly be adapted to Ruby. It runs Smalltalk, and Ruby essentially is Smalltalk. So adapting it to run Ruby is absolutely within the realm of the possible."*

Ruby e Smalltalk são parecidos, então Avi basicamente pergunta: por que não criar máquinas virtuais para Ruby aproveitando toda a tecnologia de máquinas virtuais para SmallTalk, que já têm bastante maturidade e estão no mercado a tantos anos?

Integrantes da empresa Gemstone, que possui uma das máquinas virtuais para SmallTalk mais famosas (Gemstone/S), estavam na plateia e chamaram o Avi Bryant para provar que isto era possível.

Na RailsConf de 2008, o resultado foi que a Gemstone apresentou o produto que estão desenvolvendo, conhecido como **Maglev**. É uma máquina virtual para Ruby, baseada na existente para Smalltalk. As linguagens são tão parecidas que apenas poucas instruções novas tiveram de ser inseridas na nova máquina virtual.

Os números apresentados são surpreendentes. Com tão pouco tempo de desenvolvimento, conseguiram apresentar um ganho de até 30x de performance em alguns micro benchmarks. Um dos testes mais conhecidos sobre a performance de interpretadores e máquinas virtuais Ruby feito por Antonio Cangiano em 2010 - **The Great Ruby Shootout** (http://programmingzen.com/2010/07/19/the-great-ruby-shootout-july-2010/)

- mostra que apesar de destacar-se em alguns testes, o Maglev mostra-se muito mais lento em alguns outros, além do alto consumo de memória.

Ruby Enterprise Edition

Para melhorar a performance de aplicações Rails e diminuir a quantidade de memória utilizada, **Ninh Bui**, **Hongli Lai** e **Tinco Andringa** (da Phusion) modificaram o interpretador Ruby e lançaram com o nome de **Ruby Enterprise Edition**.

As principais modificações no REE foram no comportamento do *Garbage Collector*, fazendo com que funcione com o recurso de *Copy on Write* disponível na maioria dos sistemas operacionais baseados em UNIX (Linux, Solaris, etc.).

Outra importante modificação foi na alocação de memória do interpretador, com o uso de bibliotecas famosas como `tcmalloc`. Os desenvolvedores da Phusion ofereceram as modificações (*patches*) para entrar na implementação oficial do Ruby.

A implementação oficial do Ruby versão 1.9, codinome *YARV*, adicionou algumas novas construções a linguagem em si, mas também resolveu muitos dos problema antes endereçados pela *REE*. Além disso, o suporte ao Ruby 1.8 começou a se tornar mais escasso, o que é um movimento natural relacionado a evolução da linguagem.

Como consequência de tudo isso, a própria *Phusion*, mantenedora do *REE*, anunciou o fim da manutenção do projeto. Caso queira ler mais a respeito do assunto, visite o anúncio oficial em: *http://blog.phusion.nl/2012/02/21/ruby-enterprise-edition-1-8-7-2012-02-released-end-of-life-imminent/*.

Capítulo 3

Ruby básico

"Pensa como pensam os sábios, mas fala como falam as pessoas simples"
– Aristóteles

3.1 Aprender Ruby?

Ruby on Rails é escrito em Ruby, e embora seja possível fazer aplicações inteiras sem ter um conhecimento razoável sobre Ruby, com o tempo surge a necessidade de entender mais profundamente determinado comportamento da linguagem e qual sua influência na maneira como o Rails trata determinada parte das nossas aplicações.

Conhecer bem a linguagem determinará a diferença entre *usar o Rails* e *conhecer o Rails*.

3.2 Executando código Ruby no Terminal: IRB e arquivos .rb

O **IRB** é um dos principais recursos disponíveis aos programadores Ruby. Funciona como um console/terminal, e os comandos vão sendo interpretados ao mesmo tempo em que vão sendo inseridos, de forma interativa. O `irb` avalia cada linha inserida e já mostra o resultado imediatamente.

Todos os comandos apresentados neste capítulo podem ser executados dentro do `irb`, ou em arquivos .rb. Por exemplo, poderíamos criar um arquivo chamado **teste.rb** e executá-lo com o seguinte comando:

```
ruby teste.rb
```

3.3 Variáveis, Strings e Comentários

Ao aprender uma nova linguagem de programação sempre nos deparamos com o famoso "Olá Mundo". Podemos imprimir uma mensagem com ruby utilizando pelo menos 3 opções: **puts, print e p**.

```
puts "Olá Mundo"
print "Olá Mundo"
p "Olá Mundo"
```

Mas qual a diferença entre eles? O `puts` imprime o conteúdo e pula uma linha, o `print` imprime, mas não pula linha, já o `p` chama o método `inspect` do elemento.

Comentários

Comentários em Ruby podem ser de uma linha apenas:

```
# Imprime uma mensagem
puts "Oi mundo"
```

Ou comentários de blocos:

```
=begin
    Imprime uma mensagem
=end
puts "Oi mundo"
```

> **Comentários de bloco**
>
> Nos comentários de bloco, tanto o "=begin" quanto o "=end" devem ficar no início da linha, na coluna 0 do arquivo. Os comentários de blocos não costumam ser muito usados Ruby, a própria documentação do Rails usa apenas comentários de linha.

Resultado da execução

Atenção para o modo de saída utilizado pelo Ruby e como poderemos identificar uma saída correta em nosso código:

- **Arquivo .rb:** Ao executarmos um código em um arquivo .rb, somente serão exibidas no terminal as saídas que nós especificamos com `puts` ou outro comando específico que veremos nos exemplos a seguir.
- **IRB:** Todo comando executado no IRB exibe ao menos uma linha que inicia com os caracteres =>. Essa linha indica o **retorno** da execução, como veremos nos exemplos a seguir. Caso utilizemos o `puts` no terminal, a execução resultará em mais de uma linha, sendo que a última, iniciando com => ainda representa o retorno da operação. A linha indicando o retorno `nil` significa que o código executado não tem um valor de retorno definido. Nesses casos sempre obteremos o valor `nil` que é um valor equivalente ao `null` do Java ou outras linguagens.

Palavras Reservadas do Ruby

alias and BEGIN begin break case class def defined do else elsif END end ensure false for if in module next nil not or redo rescue retry return self super then true undef unless until when while yield

3.4 Variáveis e atribuições

Um dos mais conceitos básicos em linguagens de programação é a declaração de variáveis, que é apenas uma associação entre um nome e um valor. Em Ruby, basta definirmos o nome da variável e atribuir um valor usando o sinal =:

```
ano = 1950
```

Mas qual o tipo dessa variável? A variável `ano` é do tipo `Fixnum`, um tipo do Ruby que representa números inteiros. Mas não informamos isso na declaração da variável `ano`, isso porque na linguagem Ruby não é necessária esse tipo de informação, já que o interpretador da linguagem infere o tipo da variável automaticamente durante a execução do código. Esta característica é conhecida como inferência de tipos.

GETS

O Ruby nos permite atribuir numa variável um valor digitado no console através do teclado, isso é feito através do método %gets%, mas com ele só poderemos atribuir Stringsatribuir outros tipos adiante. Exemplo:

```
irb(main):001:0> nome = gets
José
=> "José\n"
```

O \n representa um caracter de nova linha. Esse é um caracter que seu teclado manda quando é pressionada a tecla Enter.

3.5 Tipagem

Por não ser necessária a declaração do tipo, muitos pensam que a linguagem Ruby é fracamente tipada e com isso o tipo não importaria para o interpretador. No Ruby os tipos são importantes sim para a linguagem, tornando-a fortemente tipada, além disso, o Ruby também é implicitamente e dinamicamente tipada, implicitamente tipada pois os tipos são inferidos pelo interpretador, não precisam ser declarados e dinamicamente tipada porque o Ruby permite que o tipo da variável possa ser alterado durante a execução do programa, exemplo:

```
idade = 27
idade = "27"
```

O .class

Em Ruby temos a possibilidade de descobrir o tipo de uma variável utilizando o .class:

```
irb(main):001:0> num = 87
=> 87
irb(main):002:0> puts num.class
Fixnum
=> nil
```

3.6 Exercícios - Variáveis e Atribuições

1) Vamos fazer alguns testes com o Ruby, escrevendo o código e executando no terminal. O arquivo utilizado se chamará **restaurante_basico.rb**, pois nossa futura aplicação web será um sistema de qualificação de restaurantes.

- Crie um diretório no Desktop com o nome **ruby**
- Dentro deste diretório crie o arquivo **restaurante_basico.rb**

Ao final de cada exercício você pode rodar o arquivo no terminal com o comando **ruby restaurante_basico.rb**.

2) Vamos imprimir uma mensagem pedindo para o usuário digitar o nome do restaurante.

```
puts "Digite o nome do restaurante: "
```

3) Agora iremos guardar o nome do nosso restaurante em uma variável `nome`. Para isso usaremos o método **gets** para capturar o valor digitado no console.

```
puts "Digite o nome do restaurante"
nome = gets
```

4) Por fim vamos imprimir o nome do nosso restaurante.

```
puts "Digite o nome do restaurante"
nome = gets
print "Nome do restaurante: "
puts nome
```

3.7 String

Um tipo muito importante nos programas Ruby são os objetos do tipo `String`. As `Strings` literais em Ruby podem ser delimitadas por aspas simples ou aspas duplas além de outras formas especiais que veremos mais adiante.

```
irb(main):001:0> mensagem = "Olá Mundo"
=> "Olá Mundo"
irb(main):002:0> mensagem = 'Olá Mundo'
=> "Olá Mundo"
```

Mutabilidade

A principal característica das Strings em Ruby é que são mutáveis, diferente do Java, por exemplo.

```
irb(main):001:0> mensagem = "Bom dia, "
=> "Bom dia,"
irb(main):002:0> mensagem << " tudo bem?"
=> "Bom dia, tudo bem?"
irb(main):003:0> puts mensagem
Bom dia, tudo bem?
=> nil
```

O operador << é utilizado para a operação append de Strings, ou seja, para a concatenação de Strings em uma mesma instância. Já o operador + também concatena Strings mas não na mesma instância, isso quer dizer que o + gera novas Strings.

Interpolação de Strings

Lembra da nossa mensagem que fizemos no último exercício?

```
#Declaração da variável %%nome%%
print "Nome do restaurante: "
puts nome
```

Nós podemos simplificá-la, uma alternativa mais interessante para criar Strings com valor dinâmico é a interpolação:

```
#Declaração da variável %%nome%%
print "Nome do restaurante: #{nome}"
```

Basta colocar dentro da String a variável entre as chaves, em #{}. Mas fique atento, com Strings definidas com aspas simples não é possível fazer uso da interpolação, por isso prefira sempre o uso de String com aspas duplas.

Prefira sempre a interpolação ao invés da concatenação (+) ou do append (<<). É mais elegante e mais rápido.

O método capitalize

Repare que no caso do restaurante o usuário pode digitar qualquer nome, começando com ou sem letra maiúscula. Mas seria elegante um restaurante com nome que comece com letra minúscula? Para isso temos um método chamado **capitalize**, sua função nada mais é do que colocar a primeira letra de uma String em caixa alta.

```
irb(main):001:0> nome = "fasano"
=> "fasano"
irb(main):002:0> puts nome.capitalize
Fasano
=> nil
```

A chamada do método capitalize ao invés de alterar a variável para guardar o valor "Fasano", retorna uma nova String com o valor com a primeira letra em caixa alta. Você pode confirmar este comportamento imprimindo novamente o nome, logo após a chamada do método capitalize, repare que foi impresso exatamente o mesmo valor que definimos na declaração da variável.

```
nome = "fasano"
puts nome.capitalize #Fasano
puts nome #fasano
```

O método **capitalize** é uma função pura, porque não importa quantas vezes seja invocado, retornará sempre o mesmo valor e também não causa efeitos colaterais, ou seja, não altera um valor já calculado anteriormente.

Os métodos terminados em bang(!)

Mas e se quisermos que a mudança realizada pelo `capitalize` seja definitiva? Para isso acrescentamos o caracter **!** no final do método.

```
nome = "fasano"
puts nome.capitalize! #Fasano
puts nome #Fasano
```

O caracter **!** é chamado de bang e deve ser usado com moderação, já mudar definitivamente o valor onde foi utilizado.

3.8 Exercícios - Strings

1) Primeiramente, vamos alterar o modo como o nome do nosso restaurante é impresso. Abra o arquivo **restaurante_basico.rb** e passe a utilizar a concatenação.

   ```
   print "Nome do restaurante: " + nome
   ```

2) Já deu uma melhorada, só que aprendemos uma maneira mais elegante de fazer concatenação, que é a interpolação, vamos passar a utilizá-la.

   ```
   print "Nome do restaurante: #{nome}"
   ```

Default encoding

A partir de sua versão 1.9 o Ruby tem a capacidade de tratar de modo separado o encoding de arquivo (external encoding), encoding de conteúdo de arquivo (internal encoding), além do encoding aceito no IRB.

O encoding padrão é o US-ASCII ou ISO-8859-1, dependendo da plataforma, e nenhum dos dois tem suporte a caracteres acentuados e outros símbolos úteis. Outro encoding aceito é o UTF-8 (Unicode) que aceita os caracteres acentuados que necessitamos para nossas mensagens em português.

Para configurar o internal encoding de nossos arquivos, sempre devemos adicionar uma linha de comentário especial na **primeira linha** dos arquivos **.rb**:

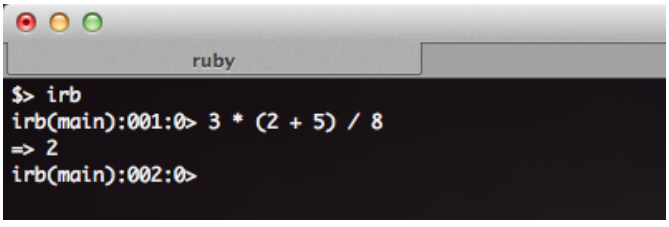
```
#coding:utf-8
puts "Olá, mundo!"
```

Para o IRB podemos configurar o encoding no comando `irb` com a opção de encoding. O comando completo é `irb -E UTF-8:UTF-8`. Em algumas plataformas é possível utilizar a forma abreviada `irb -U`.

3.9 Tipos e operações básicas

Ruby permite avaliar expressões aritméticas tradicionais:

```
irb(main):001:0> 3*(2+5)/8
=> 2
```



Em Ruby existem 3 tipos numéricos básicos: Fixnum, Bignum e Float Fixnum é o tipo principal para números inteiros.

Podemos verificar isso usando .class

```
irb(main):001:0> 99.class
=> Fixnum
```

Números inteiros muito grandes são convertidos automaticamente para Bignum, de forma a não sofrerem perda de precisão

```
irb(main):002:0> 9999999999999999999.class
=> Bignum
```

Qualquer número com casas decimais é do tipo Float

```
irb(main):003:0> 15.0.class
=> Float
```

Quando trabalhamos com números, provavelmente faremos operações matemáticas também Para isso, devemos usar os operadores apropriados. Os principais operadores aritméticos em Ruby são:

- + Soma
- - Subtração
- / Divisão
- * Multiplicação
- ** Potência
- % Modulo (retorna o resto de uma divisão)

```
irb(main):001:0> 2 + 2
=> 4
irb(main):002:0> 5 - 3
=> 2
irb(main):003:0> 10 / 2
=> 5
irb(main):004:0> 15 * 2
=> 30
irb(main):005:0> 3 ** 2
=> 9
```

Agora temos um problema: Se terntarmos fazer uma divisão cujo resultado não seja um número inteiro, este será arredondado para baixo!

```
irb(main):001:0> 15/2
=> 7
```

Pare resolver esse problema, podemos usar um Float no lugar de Fixnum, em qualquer lado da operação:

```
irb(main):002:0> 15.0 / 2
=> 7.5
```

Ranges

Ruby fornece uma maneira de trabalharmos com sequências de uma forma bem simples: (1..3) # range representando números de 1 a 3. ('a'..'z') # range representando letras minúsculas do alfabeto (0...5) # range representando números de 0 a 4.

Símbolos

Símbolos também são texto, como as Strings. Só que devem ser precedidos do carácter `':'`, ao invés de aspas e pertencem à classe `Symbol`:

```
>> puts :simbolo
simbolo
=> nil
>> :simbolo.class
=> Symbol
```

As principais diferenças são:

- São imutáveis. Uma vez criado, um símbolo não pode ser alterado. Se precisarmos de um novo valor, precisa criar um novo objeto símbolo.
- São compartilhados. Símbolos com o mesmo valor, mesmo que em locais diferentes do código, compartilham do mesmo objeto. Isto é, qualquer lugar da aplicação que contenha, por exemplo, `:writable`, se referem a um mesmo objeto do tipo `Symbol`.

Por causa destas características, símbolos são análogos às Strings do Java. As Strings do Ruby estão mais para o `StringBuilder` do Java. Por serem imutáveis e compartilhados, objetos `Symbol` geralmente são usados como identificadores e para nomenclatura (labels). Durante o curso usaremos muito este recurso.

3.10 Exercícios - Tipos

1) Em Ruby tudo é objeto. Vamos fazer alguns testes curiosos e descobrir os tipos:

```
# Tipos
3.times{
    puts "O 3 é um objeto!"
}
puts 3.class
puts 333333333333333333333.class
```

2) Podemos fazer as operações básicas em ruby de forma simples:

```
# imprime conta
puts 3*(2+5)/8
```

3.11 Estruturas de controle

Para utilizar estruturas de controle em ruby, precisamos antes conhecer os operadores booleanos, `true` e `false`. Os operadores booleanos aceitam quaisquer expressões aritméticas:

```
>> 3 > 2
=> true
>> 3+4-2 <= 3*2/4
=> false
```

Os operadores booleanos são: ==, >, <, >= e <=. Expressões booleanas podem ainda ser combinadas com os operadores `&&` (`and`) e `||` (`or`).

O `if` do ruby aceita qualquer expressão booleana, no entanto, cada objeto em Ruby possui um "valor booleano". Os únicos objetos de valor booleano `false` são o próprio `false` e o `nil`. Portanto, qualquer valor pode ser usado como argumento do `if`:

```
>> variavel = nil
=> nil
>> if(variavel)
>>   puts("so iria imprimir se variavel != null")
>> end
=> nil
>> if(3 == 3)
>>   puts("3 é 3")
>> end
3 é 3
=> nil
```

Teste também o switch:

```
def procura_sede_copa_do_mundo( ano )
   case ano
   when 1895..1993
     "Não lembro... :)"
   when 1994
     "Estados Unidos"
   when 1998
     "França"
   end
end

puts procura_sede_copa_do_mundo(1994)
```

O código acima funciona como uma série de if/elsif :

```
if 1994 == ano
   "Estados Unidos"
 elsif 1998 == ano
   "França"
 elsif 1895..1993 == ano
     "Não lembro... :)"
 end
```

Utilizar um laço de repetições pode poupar muito trabalho. Em ruby o código é bem direto:

```
for i in (1..3)
    x = i
end
```

Ruby possui bom suporte a expressões regulares, fortemente influenciado pelo Perl. Expressões regulares literais são delimitadas por / (barra).

```
>> /rio/ =~ "são paulo"
=> nil
>> /paulo/ =~ "são paulo"
=> 4
```

O operador `=~` faz a função de `match` e retorna a posição da String onde o padrão foi encontrado, ou `nil` caso a String não bata com a expressão regular. Expressões regulares aparecem com uma frequência maior em linguagens dinâmicas, e, também por sua sintaxe facilitada no Ruby, utilizaremos bastante.

Rubular.com

Um site muito interessante para testar Expressões Regulares é o http://rubular.com/

MatchData

Há também o método `match`, que retorna um objeto do tipo `MatchData`, ao invés da posição do match. O objeto retornado pelo método `match` contém diversas informações úteis sobre o resultado da expressão regular, como o valor de agrupamentos (`captures`) e posições (`offset`) em que a expressão regular bateu.

Operador ou igual

O operador ||= atribui um valor apenas a variável esteja vazia. é muito utilizado para carregar valores de maneira "lazy".

```
nome ||= "anonimo"
```

Nesse caso, se `nome` é nulo, ele será preenchido com `"anonimo"`.

3.12 Exercícios - Estruturas de controle e Regexp

1) Nosso restaurante pode ter notas boas se a mesma for superior a 7, ou ruim caso contrário.

```
# estruturas de controle: if
nota = 10

if nota > 7
    puts "Boa nota!"
else
    puts "Nota ruim!"
end
```

Teste com notas diferentes e verifique as saídas no terminal.

2) Implemente um for em ruby:

```
# estruturas de controle: for
for i in (1..3)
  x = i
end
puts x
```

3) Teste o operado de expressões regulares ""

```
puts /rio/ =~ "são paulo" # nil
puts /paulo/ =~ "são paulo" # 4
```

Abra o site http://rubular.com/ e teste outras expressões regulares!

4) O operador "||=" é considerado uma forma elegante de fazer um if. Verifique se uma variável já foi preenchida, caso não estiver, vamos atribuir um valor a ela.

```
restaurante ||= "Fogo de Chao"
puts restaurante
```

3.13 Desafios

1) Sem tentar executar o código abaixo, responda: Ele funciona? Por que?

```
resultado = 10 + 4
texto = "O valor é " + resultado
puts(texto)
```

2) E o código abaixo, deveria funcionar? Por que?

```
puts(1+2)
```

3) Baseado na sua resposta da primeira questão, por que o código abaixo funciona?

```
resultado = 10 + 3
texto = "O valor é: #{resultado}"
```

4) Qual a saída deste código?

```
resultado = 10 ** 2
puts('o resultado é: #{resultado}')
```

5) **(Para Casa)** Pesquise sobre outras maneiras de criar Strings literais em Ruby.

6) Por que a comparação entre símbolos é muito mais rápida que entre Strings?

```
simbolo1 = :abc
simbolo2 = :abc
simbolo1 == simbolo2
# => true

texto1 = "abc"
texto2 = "abc"
texto1 == texto2
# => true
```

Capítulo 4

Mais Ruby: classes, objetos e métodos

"Uma imagem vale mais que mil palavras"
– Napoleão Bonaparte

> **Orientação a objetos pura**
>
> Entre as linguagens de programação orientada a objetos, muito se discute se são puramente orientadas a objeto ou não, já que grande parte possui recursos que não se comportam como objetos.
>
> Os tipos primitivos de Java são um exemplo desta contradição, já que não são objetos de verdade. Ruby é considerada uma linguagem puramente orientada a objetos, já que **tudo** em Ruby é um objeto (inclusive as classes, como veremos).

4.1 Mundo orientado a objetos

Ruby é uma linguagem puramente orientada a objetos, bastante influenciada pelo Smalltalk. Desta forma, **tudo** em Ruby é um objeto, até mesmo os tipos básicos que vimos até agora.

Uma maneira simples de visualizar isso é através da chamada de um método em qualquer um dos objetos:

```
"strings são objetos".upcase()
:um_simbolo.object_id()
```

Até os números inteiros são objetos, da classe `Fixnum`:

```
10.class()
```

4.2 Métodos comuns

Uma das funcionalidades comuns a diversas linguagens orientadas a objeto está na capacidade de, dado um objeto, descobrir de que tipo ele é. No ruby, existe um método chamado `class()`, que retorna o tipo do objeto, enquanto `object_id()`, retorna o número da referência, ou identificador único do objeto dentro da memória heap.

Outro método comum a essas linguagens, é aquele que "transforma" um objeto em uma `String`, geralmente usado para log. O Ruby também disponibiliza esse método, através da chamada ao `to_s()`.

Adicionalmente aos tipos básicos, podemos criar nossos próprios objetos, que já vem com esses métodos que todo objeto possui (`class`, `object_id`).

Para criar um objeto em Ruby, basta invocar o método `new` na classe que desejamos instanciar. O exemplo a seguir mostra como instanciar um objeto:

```
# criando um objeto
objeto = Object.new()
```

4.3 Definição de métodos

`def` é uma palavra chave do Ruby para a **definição** (criação) de métodos, que podem, claro, receber parâmetros:

```
def pessoa.vai(lugar)
  puts "indo para " + lugar
end
```

Mas, e o retorno de um método? Como funciona? Para diminuir o excesso de código que as linguagens costumam introduzir (chamado de ruído sintático), o Ruby optou por retornar o resultado da execução da última instrução executada no método. O exemplo a seguir mostra um método que devolve uma `String`:

```
def pessoa.vai(lugar)
  "indo para " + lugar
end
```

Para visualizar esse retorno funcionando, podemos acessar o método e imprimir o retorno do mesmo:

```
puts pessoa.vai("casa")
```

Podemos ainda refatorar o nosso método para usar interpolação:

```
def pessoa.vai(lugar)
  "indo para #{lugar}"
end
```

Para receber vários argumentos em um método, basta separá-los por vírgula:

```
def pessoa.troca(roupa, lugar)
    "trocando de #{roupa} no #{lugar}"
end
```

A invocação desses métodos é feita da maneira tradicional:

```
pessoa.troca('camiseta', 'banheiro')
```

Alguns podem até ter um valor padrão, fazendo com que sejam opcionais:

```
def pessoa.troca(roupa, lugar='banheiro')
  "trocando de #{roupa} no #{lugar}"
end

# invocação sem o parametro:
pessoa.troca("camiseta")

# invocação com o parametro:
pessoa.troca("camiseta", "sala")
```

4.4 Exercícios - Métodos

1) Métodos são uma das formas com que os objetos se comunicam. Vamos utilizar os conceitos vistos no capítulo anterior porém indo um pouco mais além na orientação a objetos agora.

   - Crie mais um arquivo no seu diretório **ruby** chamado **restaurante_avancado**.

   Ao final de cada exercício você pode rodar o arquivo no terminal com o comando **ruby restaurante_avancado.rb**.

2) Vamos simular um método que atribui uma nota a um restaurante.

```
# declaração do método
def qualifica(nota)
    puts "A nota do restaurante foi #{nota}"
end

# chamada do método
qualifica(10)
```

3) Podemos ter parâmetros opcionais em ruby utilizando um valor padrão.

```
def qualifica(nota, msg="Obrigado")
    puts "A nota do restaurante foi #{nota}. #{msg}"
end

# chamadas com parâmetros opcionais
qualifica(10)
qualifica(1, "Comida ruim.")
```

4.5 Discussão: Enviando mensagens aos objetos

1) Na orientação a objetos a chamada de um método é análoga ao envio de uma mensagem ao objeto. Cada objeto pode reagir de uma forma diferente à mesma mensagem, ao mesmo estímulo. Isso é o polimorfismo.

   Seguindo a ideia de envio de mensagens, uma maneira alternativa de chamar um método é usar o método `send()`, que todo objeto em Ruby possui.

   ```
   pessoa.send(:fala)
   ```

   O método `send` recebe como argumento o nome do método a ser invocado, que pode ser um símbolo ou uma string. De acordo com a orientação a objetos é como se estivéssemos enviando a mensagem **"fala"** ao objeto `pessoa`.

   Além da motivação teórica, você consegue enxergar um outro grande benefício dessa forma de invocar métodos, através do `send()`? Qual?

4.6 Classes

Para não precisar adicionar sempre todos os métodos em todo objeto que criamos, Ruby possui classes, que atuam como fábricas (molde) de objetos. Classes possibilitam a criação de objetos já incluindo alguns métodos.

```
class Pessoa
  def fala
    puts "Sei Falar"
  end

  def troca(roupa, lugar="banheiro")
    "trocando de #{roupa} no #{lugar}"
  end
end
```

```
p = Pessoa.new
# o objeto apontado por p já nasce com os métodos fala e troca.
```

Todo objeto em Ruby possui o método `class`, que retorna a classe que originou este objeto (note que os parênteses podem ser omitidos na chamada e declaração de métodos):

```
p.class
# => Pessoa
```

O diferencial de classes em Ruby é que são abertas. Ou seja, **qualquer classe** pode ser alterada a qualquer momento na aplicação. Basta "reabrir" a classe e fazer as mudanças:

```
class Pessoa
  def novo_metodo
    # ...
  end
end
```

Caso a classe `Pessoa` já exista estamos apenas reabrindo sua definição para **adicionar** mais código. Não será criada uma nova classe e nem haverá um erro dizendo que a classe já existe.

4.7 Exercícios - Classes

1) Nosso método qualifica não pertence a nenhuma classe atualmente. Crie a classe **Restaurante** e instancie o objeto.

```
class Restaurante
    def qualifica(nota, msg="Obrigado")
        puts "A nota do restaurante foi #{nota}. #{msg}"
    end
end
```

2) Crie dois objetos, com chamadas diferentes.

```
restaurante_um = Restaurante.new
restaurante_dois = Restaurante.new

restaurante_um.qualifica(10)
restaurante_dois.qualifica(1, "Ruim!")
```

4.8 Desafio: Classes abertas

1) Qualquer classe em Ruby pode ser reaberta e qualquer método redefinido. Inclusive classes e métodos da biblioteca padrão, como `Object` e `Fixnum`.

   Podemos redefinir a soma de números reabrindo a classe `Fixnum`? Isto seria útil?

```
class Fixnum
  def +(outro)
    self - outro # fazendo a soma subtrair
  end
end
```

4.9 self

Um método pode invocar outro método do próprio objeto. Para isto, basta usar a referência especial `self`, que aponta para o para o próprio objeto. É análogo ao `this` de outras linguagens como Java e C#.

Todo método em Ruby é chamado em algum objeto, ou seja, um método é sempre uma mensagem enviada a um objeto. Quando não especificado, o destino da mensagem é sempre `self`:

```
class Conta
  def transfere_para(destino, quantia)
    debita quantia
    # mesmo que self.debita(quantia)

    destino.deposita quantia
  end
end
```

4.10 Desafio: self e o método puts

1) Vimos que todo método é sempre chamado em um objeto. Quando não especificamos o objeto em que o método está sendo chamado, Ruby sempre assume que seja em `self`.

   Como tudo em Ruby é um objeto, todas as operações devem ser métodos. Em especial, `puts` não é uma operação, muito menos uma palavra reservada da linguagem.

```
puts "ola!"
```

   Em qual objeto é chamado o método `puts`? Por que podemos chamar `puts` em qualquer lugar do programa? (dentro de classes, dentro de métodos, fora de tudo, ...)

2) Se podemos chamar `puts` em qualquer `self`, por que o código abaixo não funciona? (Teste!)

```
obj = "uma string"
obj.puts "todos os objetos possuem o método puts?"
```

3) (opcional) Pesquise onde (em que classe ou algo parecido) está definido o método `puts`. Uma boa dica é usar a documentação oficial da biblioteca padrão:

   http://ruby-doc.org

4.11 Atributos e propriedades: acessores e modificadores

Atributos, também conhecidos como variáveis de instância, em Ruby são sempre privados e começam com `@`. Não há como alterá-los de fora da classe; apenas os métodos de um objeto podem alterar os seus atributos (**encapsulamento!**).

```
class Pessoa
  def muda_nome(novo_nome)
    @nome = novo_nome
  end

  def diz_nome
    "meu nome é #{@nome}"
  end
end

p = Pessoa.new
p.muda_nome "João"
p.diz_nome

# => "João"
```

Podemos fazer com que algum código seja executado na criação de um objeto. Para isso, todo objeto pode ter um método especial, chamado de `initialize`:

```
class Pessoa
  def initialize
    puts "Criando nova Pessoa"
  end
end
Pessoa.new
# => "Criando nova Pessoa"
```

Os initializers são métodos privados (não podem ser chamados de fora da classe) e podem receber parâmetros. Veremos mais sobre métodos privados adiante.

```
class Pessoa
  def initialize(nome)
    @nome = nome
  end
end

joao = Pessoa.new("João")
```

Métodos acessores e modificadores são muito comuns e dão a ideia de propriedades. Existe uma convenção para a definição destes métodos, que a maioria dos desenvolvedores Ruby segue (assim como Java tem a convenção para *getters* e *setters*):

```
class Pessoa
  def nome # acessor
    @nome
  end

  def nome=(novo_nome)
    @nome = novo_nome
  end
end

pessoa = Pessoa.new
pessoa.nome=("José")
puts pessoa.nome
# => "José"
```

4.12 Syntax Sugar: facilitando a sintaxe

Desde o início, Matz teve como objetivo claro fazer com que Ruby fosse uma linguagem extremamente legível. Portanto, sempre que houver oportunidade de deixar determinado código mais legível, Ruby o fará.

Um exemplo importante é o modificador que acabamos de ver (`nome=`). Os parenteses na chamada de métodos são quase sempre opcionais, desde que não haja ambiguidades:

```
pessoa.nome= "José"
```

Para ficar bastante parecido com uma simples atribuição, bastaria colocar um espaço antes do `'='`. Priorizando a legibilidade, Ruby abre mão da rigidez sintática em alguns casos, como este:

```
pessoa.nome = "José"
```

Apesar de parecer, a linha acima **não é uma simples atribuição**, já que na verdade o método `nome=` está sendo chamado. Este recurso é conhecido como *Syntax Sugar*, já que o Ruby aceita algumas exceções na sintaxe para que o código fique mais legível.

A mesma regra se aplica às operações aritméticas que havíamos visto. Os números em Ruby também são objetos! Experimente:

```
10.class
# => Fixnum
```

Os operadores em Ruby são métodos comuns. Tudo em ruby é um objeto e todas as operações funcionam como envio de mensagens.

```
10.+(3)
```

Ruby tem *Syntax Sugar* para estes métodos operadores matemáticos. Para este conjunto especial de métodos, podemos omitir o ponto e trocar por um espaço. Como com qualquer outro método, os parenteses são opcionais.

4.13 Exercícios - Atributos e propriedades

1) Crie um **initialize** no Restaurante.

```
class Restaurante
    def initialize
        puts "criando um novo restaurante"
    end
    # não apague o método 'qualifica'
end
```

Rode o exemplo e verifique a chamada ao initialize.

2) Cada restaurante da nossa aplicação precisa de um nome diferente. Vamos criar o atributo e suas propriedades de acesso.

```
class Restaurante
    def initialize(nome)
        puts "criando um novo restaurante: #{nome}"
    end
    # não apague o método 'qualifica'
end

# crie dois nomes diferentes
restaurante_um = Restaurante.new("Fasano")
restaurante_dois = Restaurante.new("Fogo de Chao")
```

3) Embora nossos objetos tenham agora recebido um nome, esse nome não foi guardado em nenhuma variável.

```
class Restaurante
    def initialize(nome)
        puts "criando um novo restaurante: #{nome}"
        @nome = nome
    end

    # altere o método qualifica
    def qualifica(nota, msg="Obrigado")
        puts "A nota do #{@nome} foi #{nota}. #{msg}"
    end
end
```

4) Repare que a variável **nome** é uma variável de instância de Restaurante. Em ruby, o "@" definiu esse comportamento. Agora vamos fazer algo parecido com o atributo **nota** e suas propriedades de acesso.

```
class Restaurante
    def initialize(nome)
        puts "criando um novo restaurante: #{nome}"
        @nome = nome
    end

    # altere o método qualifica
    def qualifica(msg="Obrigado")
        puts "A nota do #{@nome} foi #{@nota}. #{msg}"
    end

    # propriedades
    def nota=(nota)
        @nota = nota
    end

    def nota
        @nota
    end

end

restaurante_um = Restaurante.new("Fasano")
restaurante_dois = Restaurante.new("Fogo de Chao")

restaurante_um.nota = 10
restaurante_dois.nota = 1
```

```
restaurante_um.qualifica
restaurante_dois.qualifica("Comida ruim.")
```

5) Seria muito trabalhoso definir todas as propriedades de acesso a nossa variáveis. Refatore a classe Restaurante para utilizar o **attr_accessor :nota** Seu arquivo final deve ficar assim:

```
class Restaurante
attr_accessor :nota

    def initialize(nome)
        puts "criando um novo restaurante: #{nome}"
        @nome = nome
    end

    def qualifica(msg="Obrigado")
        puts "A nota do #{@nome} foi #{@nota}. #{msg}"
    end

end

restaurante_um = Restaurante.new("Fasano")
restaurante_dois = Restaurante.new("Fogo de Chao")

restaurante_um.nota = 10
restaurante_dois.nota = 1

restaurante_um.qualifica
restaurante_dois.qualifica("Comida ruim.")
```

4.14 Coleções

Arrays em Ruby são instâncias da classe `Array`, não sendo simplesmente uma estrutura de dados, mas possuindo diversos métodos auxiliares que nos ajudam no dia-a-dia.

Um exemplo simples e encontrado em objetos de tipo `Array` em várias linguagens é o que permite a inclusão de elementos e `size` que nos retorna a quantidade de elementos lá dentro.

```
lista = Array.new
lista << "RR-71"
lista << "RR-75"
lista << "FJ-91"

puts lista.size
# => 3
```

A tarefa mais comum ao interagir com array é a de resgatar os elementos e para isso usamos `[]` passando um índice como parametro:

```
puts lista[1]
# => "RR-75"
puts lista[0]
# => "RR-71"
```

Muitas vezes já sabemos de antemão o que desejamos colocar dentro da nossa array e o Ruby fornece uma maneira literal de declará-la:

```
lista = [1, 2, "string", :simbolo, /$regex^/]
puts lista[2]
# => string
```

Um exemplo que demonstra uma aparição de arrays é a chamada ao método `methods`, que retorna uma array com os nomes de todos os métodos que o objeto sabe responder naquele instante. Esse método é definido na classe Object então todos os objetos o possuem:

```
cliente = "Petrobras"

puts cliente.methods
```

4.15 Exemplo: múltiplos parâmetros

Em alguns instantes desejamos receber um número arbitrário de argumentos em um método, por exemplo a lista de produtos que estamos comprando em um serviço:

```
def compra(produto1, produto2, produto3, produtoN)
end
```

Para receber um número qualquer de parâmetros usamos a sintaxe * do Ruby:

```
def compra(*produtos)
    # produtos é uma array
    puts produtos.size
end
```

Note que nesse caso, em Ruby receber uma array ou um número indeterminado de argumentos resulta no mesmo código interno ao método pois a variável produtos funciona como uma array.

A mudança ocorre na sintaxe de chamada a esse método. Na versão mostrada acima, invocamos o método compra com diversos parâmetros:

```
compra("Notebook", "Pendrive", "Cafeteira")
```

Já no caso de definir o método recebendo uma array, somos obrigados a definir uma array no momento de invocá-lo:

```
def compra(produtos)
    # produtos é uma array
    puts produtos.size
end
compra( ["Notebook", "Pendrive", "Cafeteira"] )
```

O operador * é chamado de *splat*.

4.16 Hashes

Mas nem sempre desejamos trabalhar com arrays cuja maneira de encontrar o que está lá dentro é através de um número. Por exemplo, ao ler um arquivo de configuração de um servidor desejamos saber qual a porta e se ele deve ativar o suporte ssh:

```
porta = 80
ssh = false
nome = Caelum.com.br
```

Nesse caso não desejamos acessar uma array através de inteiros (`Fixnum`), mas sim o valor correspondente as chaves "porta", "ssh" e "nome".

Ruby também tem uma estrutura indexada por qualquer objeto, onde as chaves podem ser de qualquer tipo, o que permite atingir nosso objetivo. A classe `Hash` é quem dá suporte a essa funcionalidade, sendo análoga aos objetos `HashMap`, `HashTable`, arrays indexados por `String` e dicionários de outras linguagens.

```
config = Hash.new
    config["porta"] = 80
    config["ssh"] = false
    config["nome"] = "Caelum.com.br"

    puts config.size
    # => 3

    puts config["ssh"]
    # => false
```

Por serem únicos e imutáveis, símbolos são ótimos candidatos a serem chaves em Hashes, portanto poderíamos trabalhar com:

```
config = Hash.new
    config[:porta] = 80
```

Ao invocar um método com um número maior de parâmetros, o código fica pouco legível, já que Ruby não possui tipagem explícita, apesar de esta não ser a única causa.

Imagine que tenho uma conta bancária em minhas mãos e desejo invocar o método de transferência, que requer a conta destino, a data na qual o valor será transferido e o valor. Então tento invocar:

```
aluno = Conta.new
escola = Conta.new

# Time.now gera um objeto Time que representa "agora"
aluno.transfere(escola, Time.now, 50.00)
```

No momento de executar o método descobrimos que a ordem dos parâmetros era incorreta, o valor deveria vir antes da data, de acordo com a definição do método:

```
class Conta

    def transfere(destino, valor, data)
        # executa a transferência
    end

end
```

Mas só descobrimos esse erro ao ler a definição do método na classe original, e para contornar esse problema, existe um movimento que se tornou comum com a popularização do Rails 2, passando parâmetro através de hash:

```
aluno.transfere( {:destino => escola, :data => Time.now,
    :valor => 50.00} )
```

Note que o uso do Hash implicou em uma legibilidade maior apesar de uma proliferação de palavras:

```
def transfere(argumentos)
    destino = argumentos[:destino]
    data = argumentos[:data]
    valor = argumentos[:valor]
    # executa a transferência
end
```

Isso acontece pois a semântica de cada parâmetro fica explícita para quem lê o código, sem precisar olhar a definição do método para lembrar o que eram cada um dos parâmetros.

Combinando Hashes e não Hashes

Variações nos símbolos permitem melhorar ainda mais a legibilidade, por exemplo:

```
class Conta
    def transfere(valor, argumentos)
        destino = argumentos[:para]
        data = argumentos[:em]
        # executa a transferência
    end
end

        aluno.transfere(50.00, {:para => escola, :em => Time.now})
```

Para quem utiliza Ruby 1.9, ao criar um Hash onde a chave é um símbolo, podemos utilizar uma outra sintaxe que por vezes se torna mais legível:

```
aluno.transfere(50.00, {para: escola, em: Time.now})
```

Além dos parênteses serem sempre opcionais, quando um Hash é o último parâmetro de um método, as chaves podem ser omitidas (*Syntax Sugar*).

```
aluno.transfere :destino => escola, :valor => 50.0, :data => Time.now
```

4.17 Exercícios - Arrays e Hashes

1) No seu diretório **ruby** crie mais um arquivo chamado **listas.rb**. Teste um array adicionando dois elementos.

```
nomes = []

nomes[0] = "Fasano"
nomes << "Fogo de Chao"

for nome in nomes
  puts nome
end
```

2) Vamos juntar os conhecimentos de classes e métodos para criar uma franquia onde podemos adicionar uma lista de restaurantes. Para isso teremos que criar na inicialização um array de restaurantes.

```
class Franquia

  def initialize
    @restaurantes = []
  end

  def adiciona(restaurante)
    @restaurantes << restaurante
  end

  def mostra
    for restaurante in @restaurantes
      puts restaurante.nome
    end
  end

end

class Restaurante
  attr_accessor :nome
end

restaurante_um = Restaurante.new
restaurante_um.nome = "Fasano"

restaurante_dois = Restaurante.new
restaurante_dois.nome = "Fogo de Chao"

franquia = Franquia.new
franquia.adiciona restaurante_um
franquia.adiciona restaurante_dois

franquia.mostra
```

3) O método adiciona recebe apenas um restaurante. Podemos usar a syntax com * e refatorar o método para permitir múltiplos restaurantes como parâmetro.

```
# refatore o método adiciona
  def adiciona(*restaurantes)
  for restaurante in restaurantes
    @restaurantes << restaurante
  end
end

  # adicione ambos de uma só vez
  franquia.adiciona restaurante_um, restaurante_dois
```

4) Hashes são muito úteis para passar valores identificados nos parâmetros. Na classe **restaurante** adicione o método **fechar_conta** e faça a chamada.

```
def fechar_conta(dados)
    puts "Conta fechado no valor de #{dados[:valor]}
    e com nota #{dados[:nota]}. Comentário: #{dados[:comentario]}"
end

restaurante_um.fechar_conta :valor => 50, :nota => 9,
    :comentario => 'Gostei!'
```

4.18 Blocos e Programação Funcional

Imagine o exemplo a seguir que soma o saldo das contas de um banco:

```
class Banco

    def initialize(contas)
        @contas = contas
    end

    def status
        saldo = 0
        for conta in @contas
            saldo += conta
        end
        saldo
    end

end

banco = Banco.new([200, 300, 400])
banco.status
```

Esse processo é executado em diversos pontos da nossa aplicação e todos eles precisam exatamente desse comportamento.

Em um dia ensolarado, um ponto de nossa aplicação passa a necessitar da impressão dos saldos parciais, isso é que cada soma seja impressa. Como fazer isso sem alterar os outros tantos pontos de execução e sem duplicação de código?

O primeiro passo é perceber a necessidade que temos de inserir um código novo, desejamos incluir o seguinte código:

```
for conta in @contas
    saldo += conta
    puts saldo # essa linha é nova
end
```

Então na chamada específica desse método, passemos esse código como “parâmetro":

```
banco.status do |saldo|
    puts saldo
end
```

Isso não é um parâmetro, e sim um bloco de código, o código que desejamos executar. Note que esse bloco recebe um parâmetro chamado **saldo** e que esse parâmetro é o saldo parcial que desejamos imprimir. Para facilitar a leitura podemos renomeá-lo para **saldo_parcial**:

```
banco.status do |saldo_parcial|
    puts saldo_parcial
end
```

Imagine que o bloco funciona exatamente como a definição de uma função em ruby: ele pode receber parâmetros e ser invocado. Falta invocá-lo no código do banco, para dar a chance de execução a cada chamada do laço:

```
class Banco
    # initialize...
    def status(&block)
        saldo = 0
        for conta in @contas
            saldo += conta
            block.call(saldo)
        end
        saldo
    end
end
```

Note que block é um objeto que ao ter o método `call` invocado, chamará o bloco que foi passado, concluindo nosso primeiro objetivo: dar a chance de quem se interessar no saldo parcial, fazer algo com ele.

Qualquer outro tipo de execução, como outros cálculos, que eu desejar fazer para cada saldo, posso fazê-lo passando blocos distintos.

Ainda faltou manter compatibilidade com aquelas chamadas que não possuem bloco. Para isso, basta verificarmos se foi passado algum bloco como parâmetro para nossa função, e somente nesse caso invocá-lo.

Isto é, se o bloco foi dado (`block_given?`), então invoque o bloco:

```
for conta in @contas
    saldo += conta

    if block_given?
        block.call(saldo)
    end

end
```

> **Outra sintaxe de bloco**
>
> Existe uma outra sintaxe que podemos utilizar para passar blocos que envolve o uso de chaves:
>
> ```
> banco.status { |saldo_parcial| puts saldo_parcial }
> ```

Como vimos até aqui, o método que recebe um bloco pode decidir se deve ou não chamá-lo. Para chamar o bloco associado, existe uma outra abordagem com a palavra `yield`:

```
if block_given?
    yield(saldo)
end
```

Nessa abordagem, não se faz necessário receber o argumento &block portanto o código do Banco seria:

```
class Banco
    # initialize...
    def status
        saldo = 0
        for conta in @contas
            saldo += conta
            if block_given?
                yield(saldo)
            end
        end
        saldo
    end
end
```

Entender quando usar blocos costuma parecer complicado no início, não se preocupe . Você pode enxergá-los como uma oportunidade do método delegar parte da responsabilidade a quem o chama, permitindo customizações em cada uma de suas chamadas (estratégias diferentes, ou callbacks).

A biblioteca padrão do Ruby faz alguns usos muito interessantes de blocos. Podemos analisar a forma de iterar em coleções, que é bastante influenciada por técnicas de programação funcional.

Dizer que estamos passando uma função (pedaço de código) como parâmetro a outra função é o mesmo que passar blocos na chamada de métodos.

Para iterar em uma `Array` possuímos o método `each`, que chama o bloco de código associado para cada um dos seus items, passando o item como parâmetro ao bloco:

```
lista = ["rails", "rake", "ruby", "rvm"]
lista.each do |programa|
  puts programa
end
```

A construção acima não parece trazer muitos ganhos se comparada a forma tradicional e imperativa de iterar em um array (`for`).

Imagine agora uma situação onde queremos colocar o nome de todos os funcionários em maiúsculo, isto é, aplicar uma função para todos os elementos de uma array, construindo uma array nova.

```
funcionarios = ["Guilherme", "Sergio", "David"]
nomes_maiusculos = []

for nome in funcionarios
    nomes_maiusculos << nome.upcase
end
```

Poderíamos usar o método each:

```
funcionarios = ["Guilherme", "Sergio", "David"]
nomes_maiusculos = []

funcionarios.each do |nome|
    nomes_maiusculos << nome.upcase
end
```

Mas as duas abordagens envolvem o conceito de dizer a linguagem que queremos adicionar um elemento a uma lista existente: o trabalho é imperativo.

Lembrando que o bloco, assim como uma função qualquer, possui retorno, seria muito mais compacto se pudéssemos exprimir o desejo de criar a array diretamente:

```
funcionarios = ["Guilherme", "Sergio", "David"]

nomes_maiusculos = funcionarios.cria_uma_array
```

O código dessa função deve iterar por cada elemento, adicionando eles dentro da nossa array, *exatamente* como havíamos feito antes:

```
class Array
    def cria_uma_array
        array = []
        self.each do |elemento|
            array << elemento.upcase
        end
        array
    end
end
```

Mas podemos reclamar que o método upcase nem sempre vai funcionar: a cada chamada de cria_uma_array queremos executar um comportamento diferente. E essa é a dica para utilização de blocos: passe um bloco que customiza o comportamento do método cria_uma_array:

```
nomes_maiusculos = funcionarios.cria_uma_array do |nome|
    nome.upcase
end
```

E faça o cria_uma_array invocar esse bloco:

```
class Array
    def cria_uma_array
        array = []
        self.each do |elemento|
            array << yield(elemento)
        end
    end
end
```

Esse método que criamos já existe e se chama `map` (ou `collect`), que coleta os retornos de tomas as chamadas do bloco associado:

```
funcionarios = ["Guilherme", "Sergio", "David"]

nomes_maiusculos = funcionarios.map do |nome|
    nome.upcase
end
```

Na programação imperativa tradicional precisamos de no mínimo mais duas linhas, para o array auxiliar e para adicionar os items maiúsculos no novo array.

Diversos outros métodos do módulo `Enumerable` seguem a mesma ideia: `find`, `find_all`, `grep`, `sort`, `inject`. Não deixe de consultar a documentação, que pode ajudar a criar código mais compacto.

Só tome cuidado pois código mais compacto nem sempre é mais legível e fácil de manter.

4.19 Exercícios - Blocos

1) Refatore o método mostra de **Franquia** para iterar sobre os elementos usando blocos.

```
def mostra
  @restaurantes.each do |r|
    puts r.nome
  end
end
```

2) Crie um método **relatorio** que envia a lista de restaurantes da franquia. Repare que esse método não sabe o que ocorrerá com cada item da lista. Ele disponibiliza os itens e é o bloco passado que define o que fazer com eles.

```
def relatorio
  @restaurantes.each do |r|
    yield r
  end
end

  # chamada com blocos
  franquia.relatorio do |r|
    puts "Restaurante cadastrado: #{r.nome}"
  end
```

4.20 Para saber mais: Mais sobre blocos

Analise e rode o código abaixo, olhe na documentação o método **sort_by** e teste o **next**.

```
caelum = [
   {:ruby => 'rr-71', :java => 'fj-11'},
   {:ruby => 'rr-75', :java => 'fj-21'}
 ]

caelum.sort_by { |curso| curso[:ruby] }.each do |curso|
   puts "Curso de RoR na Caelum: #{ curso[:ruby] }"
 end
```

```
caelum.sort_by { |curso| curso[:ruby] }.each do |curso|
   next if curso[:ruby] == 'rr-71'
   puts "Curso de RoR na Caelum: #{ curso[:ruby] }"
end
```

4.21 Desafio: Usando blocos

1) Queremos imprimir o nome de todos os alunos de uma turma com o código a seguir:

```
fj91 = Turma.new("Guilherme", "Paulo", "Paniz")

fj91.each do |nome|
    puts nome
end
```

Crie a sua classe **Turma** que durante a invocação do método each, itera por todos os nomes passados em seu construtor.

Tente lembrar dos conceitos de blocos e programação funcional para resolver.

A biblioteca padrão do Ruby faz usos bastante interessante de módulos como mixins. Os principais exemplos são os módulos `Enumerable` e `Comparable`.

4.22 Manipulando erros e exceptions

Uma *exception* é um tipo especial de objeto que estende ou é uma instância da classe `Exception`. *Lançar* uma exception significa que algo não esperado ou errado ocorreu no fluxo do programa. *Raising* é a palavra usada em ruby para lançamento de exceptions. Para tratar uma exception é necessário criar um código a ser executado caso o programa receba o erro. Para isso existe a palavra `rescue`.

Exceptions comuns

A lista abaixo mostra as exceptions mais comuns em ruby e quando são lançadas, todas são filhas de `Exception`. Nos testes seguintes você pode usar essas exceptions.

- **RuntimeError** : É a exception padrão lançada pelo método `raise`.
- **NoMethodError** : Quando um objeto recebe como parametro de uma mensagem um nome de método que não pode ser encontrado.
- **NameError** : O interpretador não encontra uma variável ou método com o nome passado.
- **IOError** : Causada ao ler um stream que foi fechado, tentar escrever em algo *read-only* e situações similares.

- **Errno::error** : É a família dos erros de entrada e saída (IO).
- **TypeError** : Um método recebe como argumento algo que não pode tratar.
- **ArgumentError** : Causada por número incorreto de argumentos.

4.23 Exercício: Manipulando exceptions

1) Em um arquivo ruby crie o código abaixo para testar exceptions. O método `gets` recupera o valor digitado. Teste também outros tipos de exceptions e digite um número inválido para testar a exception.

```
print "Digite um número:"
numero = gets.to_i

begin
    resultado = 100 / numero
rescue
    puts "Número digitado inválido!"
    exit
end

puts "100/#{numero} é #{resultado} "
```

2) (opcional) Exceptions podem ser lançadas com o comando `raise`. Crie um método que lança uma exception do tipo `ArgumentError` e capture-a com `rescue`.

```
def verifica_idade(idade)
  unless idade > 18
      raise ArgumentError,
      "Você precisa ser maior de idade para jogar jogos violentos."
    end
end

verifica_idade(17)
```

3) (opcional) É possível utilizar sua própria exception criando uma classe e estendendo de `Exception`.

```
class IdadeInsuficienteException < Exception
end

def verifica_idade(idade)
    raise IdadeInsuficienteException,
     "Você precisa ser maior de idade..." unless idade > 18
end
```

Para testar o rescue dessa exception invoque o método com um valor inválido:

```
begin
        verifica_idade(13)
rescue IdadeInsuficienteException => e
    puts "Foi lançada a exception: #{e}"
end
```

Para saber mais: Throw e catch

Ruby possui também throw e catch que podem ser utilizados com símbolos e a sintaxe lembra a de Erlang, onde catch é uma função que, se ocorrer algum throw com aquele label, retorna o valor do throw atrelado:

```
def pesquisa_banco(nome)
    if nome.size<10
        throw :nome_invalido, "Nome invalido, digite novamente"
    end
    # executa a pesquisa
    "cliente #{nome}"
end

def executa_pesquisa(nome)
    catch :nome_invalido do
        cliente = pesquisa_banco(nome)
        return cliente
    end
end

puts executa_pesquisa("ana")
# => "Nome invalido, digite novamente"

puts executa_pesquisa("guilherme silveira")
# => cliente guilherme silveira
```

4.24 Arquivos com código fonte ruby

Todos os arquivos fonte, contendo código Ruby devem ter a extensão *.rb* para que o interpretador seja capaz de carregá-lo.

Existem exceções onde a leitura dos arquivos é feita por outras bibliotecas e, nesses casos, o arquivo possui outras extensões.

Para organizar seu código, é natural dividi-lo em vários arquivos e diretórios, bastando usar o método `require` para incluir o fonte de outro arquivo.

Imagine o arquivo conta.rb:

```
class Conta

    attr_reader :saldo

    def initialize(saldo)
        @saldo = saldo
    end
end
```

Agora podemos acessar uma conta bastando primeiro importar o arquivo que a define:

```
require 'conta'

puts Conta.new(500).saldo
```

Caso a extensão *.rb* seja omitida, a extensão adequada será usada (.rb, .so, .class, .dll, etc). O Ruby procura pelo arquivo em alguns diretórios predefinidos (*Ruby Load Path*), incluindo o diretório atual.

Assim como qualquer outra linguagem isso resulta em um possível Load Hell, onde não sabemos exatamente de onde nossos arquivos estão sendo carregados. Tome bastante cuidado para a configuração de seu ambiente.

Caminhos relativos ou absolutos podem ser usados para incluir arquivos em outros diretórios, sendo que o absoluto não é recomendado devido ao atrelamento claro com uma estrutura fixa que pode não ser encontrada ao portar o código para outra máquina:

```
require 'modulo/funcionalidades/coisa_importante'
require '/usr/local/lib/my/libs/ultra_parser'
```

> **Constantes do sistema**
>
> A constante `$:`, ou `$LOAD_PATH` contém diretórios do **Load Path**:
>
> ```
> $:
> # => ["/Library/Ruby/Site/1.8", ..., "."]
> ```
>
> Existem diversas outras constantes que começam com `$`, todas elas que são resquícios de PERL e que, em sua grande maioria dentro do ambiente Rails, possuem alternativas mais compreensíveis como LOAD_PATH.

O comando `require` carrega o arquivo apenas uma vez. Para executar a interpretação do conteúdo do arquivo diversas vezes, utilize o método `load`.

```
load 'conta.rb'
load 'conta.rb'
# executado duas vezes!
```

Portanto no irb (e em Ruby em geral) para recarregar um arquivo que foi alterado é necessário executar um load e não um require - que não faria nada.

4.25 Para saber mais: um pouco de IO

Para manipular arquivos de texto existe a classe `File`, que permite manipulá-los de maneira bastante simples, utilizando blocos:

```
print "Escreva um texto: "
texto = gets
File.open( "caelum.txt", "w" ) do |f|
  f << texto
end
```

E para imprimir seu conteúdo:

```
Dir.entries('caelum').each do |file_name|
   idea = File.read( file_name )
   puts idea
end
```

Podemos lidar de maneira similar com requisições HTTP utilizando o código abaixo e imprimir o conteúdo do resultado de uma requisição:

```
require 'net/http'
Net::HTTP.start( 'www.caelum.com.br', 80 ) do |http|
    print( http.get( '/' ).body )
end
```

Capítulo 5

Metaprogramação

"A ciência nunca resolve um problema sem criar mais dez"
– George Bernard Shaw

5.1 Métodos de Classe

Classes em Ruby **também são objetos**:

```
Pessoa.class
# => Class

c = Class.new
instancia = c.new
```

Variáveis com letra maiúscula representam constantes em Ruby, que até podem ser modificadas, mas o interpretador gera um *warning*. Portanto, `Pessoa` é apenas uma constante que aponta para um objeto do tipo `Class`.

Se classes são objetos, podemos definir métodos de classe como em qualquer outro objeto:

```
class Pessoa
  # ...
end

def Pessoa.pessoas_no_mundo
  100
end
```

```
Pessoa.pessoas_no_mundo
# => 100
```

Há um *idiomismo* para definir os métodos de classe dentro da própria definição da classe, onde `self` aponta para o próprio objeto classe.

```
class Pessoa
  def self.pessoas_no_mundo
    100
  end

  # ...
end
```

5.2 Para saber mais: Singleton Classes

A definição **class << object** define as chamadas singleton classes em ruby. Por exemplo, uma classe normal em ruby poderia ser:

```
class Pessoa
  def fala
        puts 'oi'
    end
end
```

Podemos instanciar e invocar o método normalmente:

```
p = Pessoa.new
p.fala   # imprime 'oi'
```

Entretanto, também é possível definir métodos apenas para esse objeto "p", pois tudo em ruby, até mesmo as classes, são objetos, fazendo:

```
def p.anda
  puts 'andando'
end
```

O método "anda" é chamado de singleton method do objeto "p".

Um singleton method "vive" em uma singleton class. Todo objeto em ruby possui 2 classes:

- a classe a qual foi instanciado

- sua singleton class

A singleton class é exclusiva para guardar os métodos desse objeto, sem compartilhar com outras instâncias da mesma classe.

Existe uma notação especial para definir uma singleton class:

```
class << Pessoa
    def anda
        puts 'andando'
    end
end
```

Definindo o código dessa forma temos o mesmo que no exemplo anterior, porém definindo o método anda explicitamente na singleton class.É possível ainda definir tudo na mesma classe:

```
class Pessoa
    class << self
        def anda
            puts 'andando'
        end
    end
end
```

Mais uma vez o método foi definido apenas para um objeto, no caso , o objeto "Pessoa", podendo ser executado com:

```
Pessoa.anda
```

5.3 Exercícios - Ruby Object Model

1) Crie um novo arquivo chamado **metaprogramacao.rb**. Independente da instância seria interessante saber o número total de restaurantes criados. Na mesma classe restaurante, crie uma variável de classe chamada **total**.

```
class Restaurante
    def initialize(nome)
        puts "criando um novo restaurante: #{nome}"
        @@total ||= 0
        @@total = @@total + 1
        puts "Restaurantes criados: #{@@total}"
        @nome = nome
    end
end
```

Execute novamente e analise o resultado.

2) Crie um método **self** chamado **relatorio** que mostra a quantidade total de restaurantes instanciados. (Opcional: Com esse método, você pode limpar um pouco o initialize).

```
def self.relatorio
  puts "Foram criados #{@@total} restaurantes"
end

    # Faça mais uma chamada
    Restaurante.relatorio
```

> **Método de Classe é diferente de static do Java**
>
> Se você trabalha com **java** pode confundir o **self** com o **static**. Cuidado! O método definido como self roda apenas na classe, não funciona nas instâncias. Você pode testar fazendo:
>
> ```
> Restaurante.relatorio
> restaurante_um.relatorio
> ```
>
> A invocação na instância dará um: NoMethodError: undefined method 'relatorio' for #<Restaurante:0x100137b48 @nome="Fasano", @nota=10>

3) Caso sua classe possua muitos métodos de classe, é recomendado agrupá-los. Muitos códigos ruby utilizaram essa técnica, inclusive no próprio Rails Sua classe ficará então assim:

```
class Restaurante
   # initialize, qualifica ...

class << self
  def relatorio
    puts "Foram criados #{@@total} restaurantes"
  end
end

end
```

Execute e veja que o comportamento é o mesmo.

5.4 Convenções

Métodos que retornam booleanos costumam terminar com ?, para que pareçam perguntas aos objetos:

```
texto = "nao sou vazio"
texto.empty? # => false
```

Já vimos esta convenção no método `respond_to?`.

Métodos que tem efeito colateral (alteram o estado do objeto, ou que costumem lançar exceções) geralmente terminam com `!` (bang):

```
conta.cancela!
```

A comparação entre objetos é feita através do método == (sim, é um método!). A versão original do método apenas verifica se as referências são iguais, ou seja, se apontam para os mesmos objetos. Podemos reescrever este comportamento e dizer como comparar dois objetos:

```
class Pessoa
  def ==(outra)
    self.cpf == outra.cpf
  end
end
```

Na definição de métodos, procure sempre usar os parênteses. Para a chamada de métodos, não há convenção. Prefira o que for mais legível.

Nomes de variável e métodos em Ruby são sempre minúsculos e separados por `'_'` (underscore). Variáveis com nomes maiúsculo são sempre constantes. Para nomes de classes, utilize as regras de **CamelCase**, afinal nomes de classes são apenas constantes.

5.5 Polimorfismo

Ruby também tem suporte a herança simples de classes:

```
class Animal
  def come
    "comendo"
  end
end

class Pato < Animal
  def quack
    "Quack!"
  end
end

pato = Pato.new
pato.come # => "comendo"
```

Classes filhas herdam todos os métodos definidos na classe mãe.

A tipagem em Ruby não é explícita, por isso não precisamos declarar quais são os tipos dos atributos. Veja este exemplo:

```
class PatoNormal
  def faz_quack
    "Quack!"
  end
end

class PatoEstranho
  def faz_quack
    "Queck!"
  end
end

class CriadorDePatos
  def castiga(pato)
    pato.faz_quack
  end
end

pato1 = PatoNormal.new
pato2 = PatoEstranho.new
c = CriadorDePatos.new
c.castiga(pato1) # => "Quack!"
c.castiga(pato2) # => "Queck!"
```

Para o criador de patos, não interessa que objeto será passado como parâmetro. Para ele basta que o objeto saiba fazer *quack*. Esta característica da linguagem Ruby é conhecida como *Duck Typing*.

"If it walks like a duck and quacks like a duck, I would call it a duck."

5.6 Exercícios - Duck Typing

1) Para fazer alguns testes de herança e polimorfismo abra o diretório **ruby** e crie um novo arquivo chamado **duck_typing.rb**. Nesse arquivo faça o teste do restaurante herdando um método da classe **Franquia**.

```
class Franquia
    def info
        puts "Restaurante faz parte da franquia"
    end
end
```

```
class Restaurante < Franquia
end

restaurante = Restaurante.new
restaurante.info
```

Execute o arquivo no terminal e verifique a utilização do método herdado.

2) Podemos em ruby fazer a sobrescrita do método e invocar o método da classe mãe.

```
class Franquia
    def info
        puts "Restaurante faz parte da franquia"
    end
end

class Restaurante < Franquia
    def info
        super
        puts "Restaurante Fasano"
    end
end

restaurante = Restaurante.new
restaurante.info
```

3) Como a linguagem ruby é implicitamente tipada, não temos a garantia de receber um objeto do tipo que esperamos. Isso faz com que acreditamos que o objeto é de um tipo específico caso ele possua um método esperado. Faça o teste abaixo:

```
class Franquia
    def info
        puts "Restaurante faz parte da franquia"
    end
end

class Restaurante < Franquia
    def info
        super
        puts "Restaurante Fasano"
    end
end

# metodo importante
# recebe franquia e invoca o método info
def informa(franquia)
```

```
    franquia.info
  end

  restaurante = Restaurante.new
  informa restaurante
```

5.7 Modulos

Modulos podem ser usados como namespaces:

```
module Caelum
  module Validadores

    class ValidadorDeCpf
      # ...
    end

    class ValidadorDeRg
      # ...
    end

  end
end

validador = Caelum::Validadores::ValidadorDeCpf.new
```

Ou como **mixins**, conjunto de métodos a ser incluso em outras classes:

```
module Comentavel
  def comentarios
    @comentarios ||= []
  end

  def recebe_comentario(comentario)
    self.comentarios << comentario
  end
end

class Revista
  include Comentavel
  # ...
end
```

```
revista = Revista.new
revista.recebe_comentario("muito ruim!")
puts revista.comentarios
```

5.8 Metaprogramação

Por ser uma linguagem dinâmica, Ruby permite adicionar outros métodos e operações aos objetos em tempo de execução.

Imagine que tenho uma pessoa:

```
pessoa = Object.new()
```

O que aconteceria, se eu tentasse invocar um método inexistente nesse objeto? Por exemplo, se eu tentar executar

```
pessoa.fala()
```

O interpretador retornaria com uma mensagem de erro uma vez que o método não existe.

Mas e se eu desejasse, **em tempo de execução**, adicionar o comportamento (ou método) `fala` para essa pessoa. Para isso, tenho que **definir** que uma **pessoa** possui o método **fala**:

```
pessoa = Object.new()

def pessoa.fala()
  puts "Sei falar"
end
```

Agora que tenho uma pessoa com o método `fala`, posso invocá-lo:

```
pessoa = Object.new()

def pessoa.fala()
  puts "Sei falar"
end

pessoa.fala()
```

Tudo isso é chamado **meta-programação**, um recurso muito comum de linguagens dinâmicas. Meta-programação é a capacidade de gerar/alterar código em tempo de execução. Note que isso é muito diferente de um gerador de código comum, onde geraríamos um código fixo, que deveria ser editado na mão e a aplicação só rodaria esse código posteriormente.

Levando o dinamismo de Ruby ao extremo, podemos criar métodos que definem métodos em outros objetos:

```
class Aluno
  # nao sabe nada
end

class Professor
  def ensina(aluno)
    def aluno.escreve
      "sei escrever!"
    end
  end
end

juca = Aluno.new
juca.respond_to? :escreve
# => false

professor = Professor.new
professor.ensina juca
juca.escreve
# => "sei escrever!"
```

A criação de métodos acessores é uma tarefa muito comum no desenvolvimento orientado a objetos. Os métodos são sempre muito parecidos e os desenvolvedores costumam usar recursos de geração de códigos das IDEs para automatizar esta tarefa.

Já vimos que podemos criar código Ruby que escreve código Ruby (métodos). Aproveitando essa possibilidade do Ruby, existem alguns métodos de classe importantes que servem apenas para criar alguns outros métodos nos seus objetos.

```
class Pessoa
  attr_accessor :nome
end

p = Pessoa.new
p.nome = "Joaquim"
puts p.nome
# => "Joaquim"
```

A chamada do método de classe `attr_acessor`, define os métodos `nome` e `nome=` na classe `Pessoa`.

A técnica de *código gerando código* é conhecida como **metaprogramação**, ou **metaprogramming**, como já definimos.

Outro exemplo interessante de metaprogramação é como definimos a visibilidade dos métodos em Ruby. Por padrão, todos os métodos definidos em uma classe são públicos, ou seja, podem ser chamados por qualquer um.

Não existe nenhuma palavra reservada (*keyword*) da linguagem para mudar a visibilidade. Isto é feito com um método de classe. Toda classe possui os métodos `private`, `public` e `protected`, que são métodos que alteram outros métodos, mudando a sua visibilidade (código alterando código == **metaprogramação**).

Como visto, por padrão todos os métodos são públicos. O método de classe `private` altera a visibilidade de todos os métodos definidos após ter sido chamado:

```
class Pessoa

  private

  def vai_ao_banheiro
    # ...
  end
end
```

Todos os métodos após a chamada de `private` são privados. Isso pode lembrar um pouco C++, que define regiões de visibilidade dentro de uma classe (seção pública, privada, ...). Um método privado em Ruby **só pode ser chamado em self** e o `self` deve ser **implícito**. Em outras palavras, não podemos colocar o `self` explicitamente para métodos privados, como em `self.vai_ao_banheiro`.

Caso seja necessário, o método `public` faz com que os métodos em seguida voltem a ser públicos:

```
class Pessoa

  private
  def vai_ao_banheiro
    # ...
  end

  public
  def sou_um_metodo_publico
    # ...
  end
end
```

O último modificador de visibilidade é o `protected`. Métodos protected só podem ser chamados em self (implícito ou explícito). Por isso, o `protected` do Ruby acaba sendo semelhante ao protected do Java e C++, que permitem a chamada do método na própria classe e em classes filhas.

5.9 Exercícios - Metaprogramação

1) Alguns restaurantes poderão receber um método a mais chamado "cadastrar_vips". Para isso precisaremos abrir em tempo de execução a classe Restaurante.

```
# faça a chamada e verifique a exception NoMethodError
  restaurante_um.cadastrar_vips

  # adicione esse método na classe Franquia
  def expandir(restaurante)
  def restaurante.cadastrar_vips
    puts "Restaurante #{self.nome} agora com área VIP!"
  end
end

  # faça a franquia abrir a classe e adicionar o método
  franquia.expandir restaurante_um
  restaurante_um.cadastrar_vips
```

2) Um método útil para nossa franquia seria a verificação de nome de restaurantes cadastrados. vamos fazer isso usando **method_missing**.

```
# Adicione na classe Franquia
  def method_missing(name, *args)
  @restaurantes.each do |r|
    return "O restaurante #{r.nome} já foi cadastrado!"
                  if r.nome.eql? *args
  end
  return "O restaurante #{args[0]} não foi cadastrado ainda."
end

  # Faça as chamadas e analise os resultados
puts franquia.já_cadastrado?("Fasano")
  puts franquia.já_cadastrado?("Boteco")
```

Capítulo 6

Ruby on Rails

"A libertação do desejo conduz à paz interior"
– Lao-Tsé

Aqui faremos um mergulho no Rails e, em um único capítulo, teremos uma pequena aplicação pronta: com banco de dados, interface web, lógica de negócio e todo o necessário para um CRUD, para que nos capítulos posteriores possamos ver cada parte do Rails com detalhes e profundidade, criando uma aplicação bem mais completa.

6.1 Ruby On Rails

David Heinemeier Hansson criou o Ruby on Rails para usar em um de seus projetos na *37signals*, o Basecamp. Desde então, passou a divulgar o código e incentivar o uso do mesmo, e em 2006 começou a ganhar muita atenção da comunidade de desenvolvimento Web.

O Rails foi criado pensando na praticidade que ele proporcionaria na hora de escrever os aplicativos para Web. No Brasil a Caelum vem utilizando o framework desde 2007, e grandes empresas como Abril e Locaweb adotaram o framework em uma grande quantidade de projetos.

Outro atrativo do framework é que, comparado a outros, ele permite que as funcionalidades de um sistema possam ser implementadas de maneira incremental por conta de alguns padrões e conceitos adotados. Isso tornou o Rails uma das escolhas óbvias para projetos e empresas que adotam metodologias ágeis de desenvolvimento e gerenciamento de projeto.

Para saber mais sobre metodologias ágeis, a Caelum oferece os cursos *Gerenciamento ágil de projetos de Software com Scrum* (PM-83) e *Práticas ágeis de desenvolvimento de Software* (PM-87).

Como pilares do Rails estão os conceitos de *Don't Repeat Yourself* (DRY), e *Convention over Configuration* (CoC).

O primeiro conceito nos incentiva a fazer bom uso da **reutilização de código**, que é também uma das principais vantagens da orientação a objetos. Além de podermos aproveitar as características de OO do Ruby, o próprio framework nos incentiva a adotar padrões de projeto mais adequados para essa finalidade.

O segundo conceito nos traz o benefício de poder escrever muito menos código para implementar uma determinada funcionalidade em nossa aplicação, desde que respeitemos alguns padrões de nome e localização de arquivos, nome de classes e métodos, entre outras regras simples e fáceis de serem memorizadas e seguidas.

É possível, porém, contornar as exceções com algumas linhas de código a mais. No geral nossas aplicações apresentam um código bem simples e enxuto por conta desse conceito.

MVC

A arquitetura principal do Rails é baseada no conceito de separar tudo em três camadas: o **MVC** (*Model, View, Controller*). MVC é um padrão arquitetural onde os limites entre seus modelos, suas lógicas e suas visualizações são bem definidos, sendo muito mais simples fazer um reparo, uma mudança ou uma manutenção, já que essas três partes de comunicam de maneira bem desacoplada.

Meta-Framework

Rails não é baseado num único padrão, mas sim um conjunto de padrões, alguns dos quais discutiremos durante o curso.

Outros frameworks que faziam parte do núcleo do Rails antigamente foram removidos desse núcleo para diminuir o acoplamento com ele e permitir que vocês os substitua sem dificuldade, mas continuam funcionando e sendo usados em conjunto. Aqui estão alguns deles:

- ActionMailer
- ActionPack
  - Action View
  - Action Controller
- ActiveRecord
- ActiveSupport

Projetos que usam o Ruby on Rails

Há uma extensa lista de aplicações que usam o Ruby on Rails e, entre elas estão o Twitter, YellowPages, Groupon, Typo (blog open source) e o Spokeo (ferramenta de agrupamento de sites de relacionamentos).

Uma lista de sites usando Ruby on Rails com sucesso pode ser encontrada nesses endereços:

- http://rubyonrails.org/applications
- http://www.rubyonrailsgallery.com
- http://www.opensourcerails.com

6.2 Ambiente de Desenvolvimento

Por ser uma aplicação em Ruby -- contendo arquivos .rb e alguns arquivos de configuração diversos, todos baseados em texto puro -- o Ruby on Rails não requer nenhuma ferramenta avançada para que possamos criar aplicações. Utilizar um editor de textos simples ou uma IDE (*Integrated Development Environment*) cheia de recursos e atalhos é uma decisão de cada desenvolvedor.

Algumas IDEs podem trazer algumas facilidades como execução automática de testes, *syntax highlighting*, integração com diversas ferramentas, wizards, servidores, autocomplete, logs, perspectivas do banco de dados etc. Existem diversas IDEs para trabalhar com Rails, sendo as mais famosas:

- RubyMine - baseada no IntelliJ IDEA
- Aptana Studio 3 - antes conhecida como RadRails, disponível no modelo *stand-alone* ou como plug-in para Eclipse
- Ruby in Steel - para Visual Studio
- NetBeans

Segundo enquetes em listas de discussões, a maioria dos desenvolvedores Ruby on Rails não utiliza nenhuma IDE, apenas um bom **editor de texto** e um pouco de treino para deixá-lo confortável no uso do **terminal de comando** do sistema operacional é suficiente para ser bem produtivo e ainda por cima não depender de um Software grande e complexo para que você possa desenvolver.

Durante nosso curso iremos utilizar o editor de textos padrão do Ubuntu, o GEdit. Alguns outros editores são bem conhecidos na comunidade:

- TextMate - o editor preferido da comunidade Rails, somente para Mac;
- SublimeText 2 - poderoso e flexível, é compatível com plugins do TextMate. Versões para Windows, Mac OS e Linux;

- NotePad++ - famoso entre usuários de Windows, traz alguns recursos interessantes e é bem flexível;
- SciTE - editor simples, compatível com Windows, Mac e Linux;
- Vi / Vim - editor de textos poderoso, pode ser bem difícil para iniciantes. Compatível com Windows, Mac e Linux;
- Emacs - o todo poderoso editor do mundo Linux (também com versões para Windows e Mac);

6.3 Criando um novo projeto Rails

Quando instalamos a gem **rails** em nossa máquina, ganhamos o comando que iremos utilizar para gerar os arquivos inciais de nossa aplicação. Como a maioria dos comandos de terminal, podemos passar uma série de informações para que o projeto gerado seja customizado.

Para conhecer mais sobre as opções disponíveis, podemos imprimir a ajuda no console. Para isso basta executar o seguinte comando:

```
rails --help
```

Existem algumas opções de ambiente que são muito úteis quando temos diversas versões de Ruby instaladas na mesma máquina e queremos utilizar uma versão específica em nossa aplicação. Também temos uma série de opções para gerar nossa aplicação sem as ferramentas padrão (ou com ferramentas alternativas) para JavaScript, testes, pré-processamento de CSS e etc.

Como o Rails por padrão já traz um *set* de ferramentas bem pensadas para nossa produtividade, podemos nos ater ao comando básico para criação de uma aplicação:

```
rails new [caminho-da-app]
```

Caso a pasta apontada como caminho da aplicação não exista, ela será criada pelo gerador do Rails. Caso a pasta exista e contenha arquivos que possam conflitar com os gerados pelo Rails será necessário informar a ação adequada (sobrescrever, não sobrescrever) para cada um deles durante o processo de geração de arquivos.

```
rails new meu_projeto
```

Dentre todas as opções, uma das mais úteis é a que prepara nossa aplicação para conexão com um banco de dados. Como o Rails é compatível com uma grande quantidade de bancos de dados podemos escolher aquele que temos mais familiaridade, ou um que já estiver instalado em nossa máquina. Por padrão o Rails usa o SQLite3 pois é bem simples, as informações podem ser salvas em qualquer local do sistema de arquivos (por padrão dentro da aplicação) e está disponível para Windows, Mac e Linux.

Caso queira utilizar, por exemplo, o MySQL, basta passar a opção:

```
rails new meu_projeto -d mysql
```

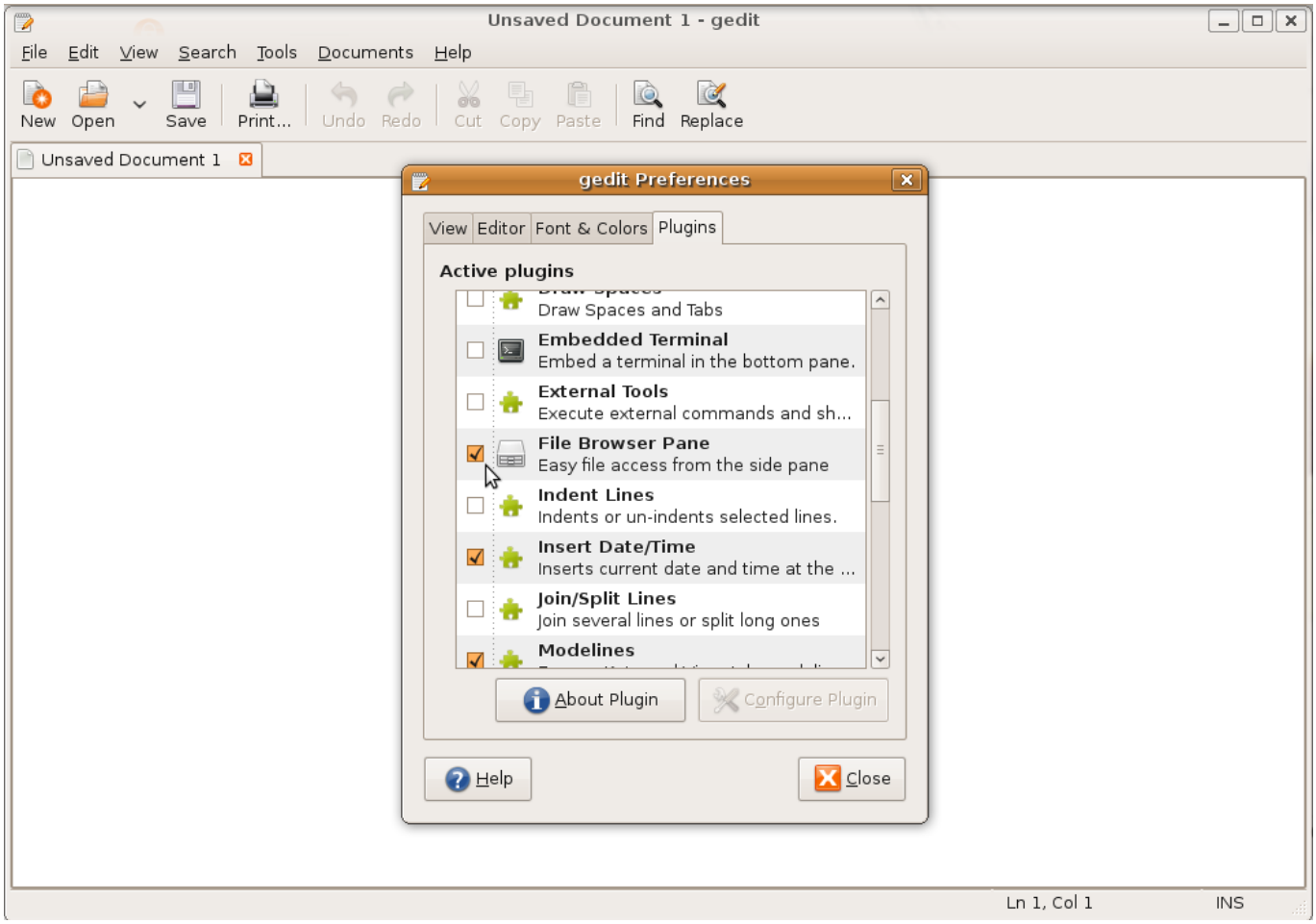
6.4 Exercícios - Iniciando o projeto

1) Inicie o GEdit:

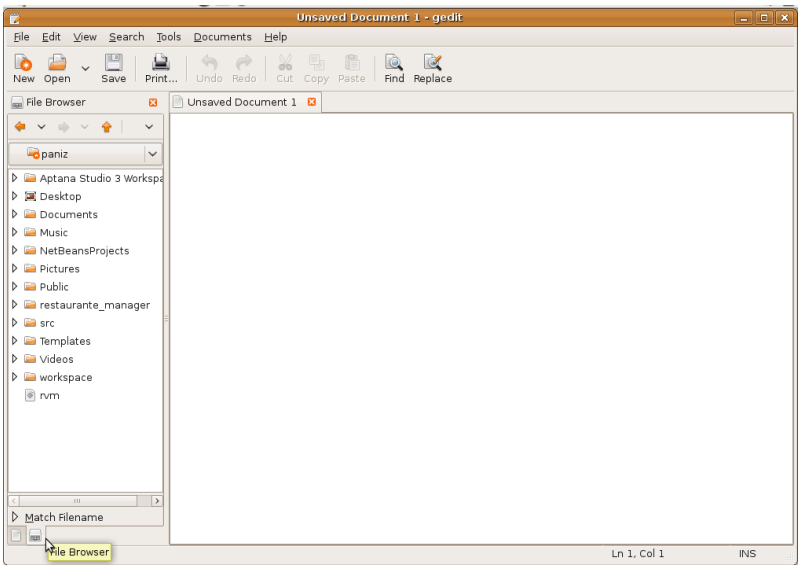
   a) Abra o GEdit através do link "Text Editor" no seu desktop ou barra de ferramentas

   b) Clique no menu **Edit** e escolher a opção **Preferences**

   c) Na nova janela clicar na aba **Plugins** e selecionar **File Browser Pane**



   d) Clicar no menu View e então selecionar **Side Pane**



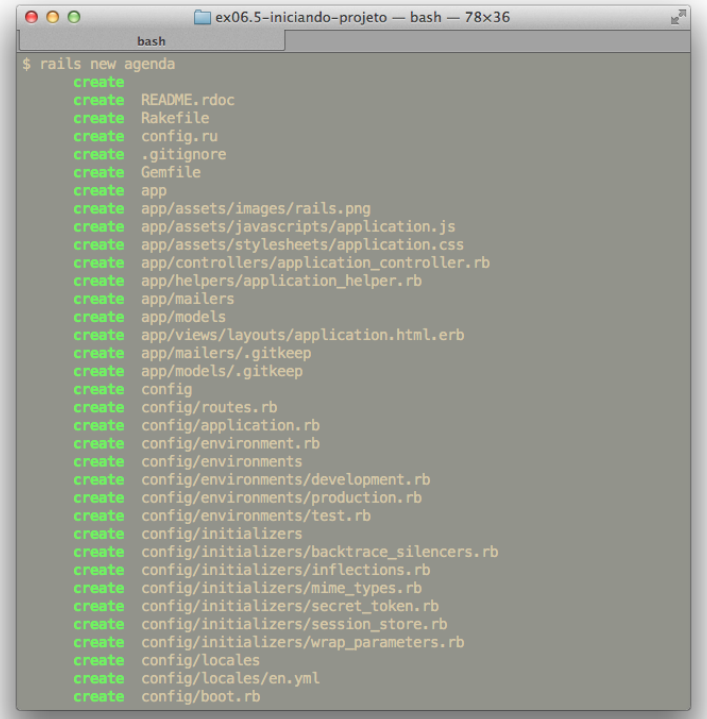
2) Inicie um novo projeto:

   a) Abra o terminal

   b) Execute o seguinte comando

   ```
   rails new agenda
   ```

   O script informa no terminal quais arquivos foram criados para nossa aplicação e logo na sequência executa o comando para instalação das dependências (bundle install).



6.5 Estrutura dos diretórios

Cada diretório tem uma função específica e bem clara na aplicação:

- **app** - A maioria dos arquivos específicos da nossa aplicação ficam aqui (inclusive todo o MVC, dividido em diretórios);
- **config** - Configurações da aplicação;

- **db** - Migrações, esquema e outros arquivos relacionados ao banco de dados;
- **doc** - Documentação do sistema;
- **lib** - Bibliotecas auxiliares;
- **log** - Informações de log;
- **public** - Arquivos estáticos que serão servidos pela WEB;
- **test** - Testes da nossa aplicação;
- **tmp** - Arquivos temporários como cache e informações de sessões;
- **vendor** - Dependências e bibliotecas de terceiros.

6.6 O Banco de dados

Uma das características do Rails é a facilidade de se comunicar com o banco de dados de modo transparente ao programador.

Um exemplo dessa facilidade é que ao gerar a aplicação, ele criou também um arquivo de configuração do banco, pronto para se conectar. O arquivo está localizado em **config/database.yml**. Como o banco de dados padrão é o SQLite3 -- por ser pequeno, versátil e comportar bem um ambiente de desenvolvimento -- a configuração foi gerada para ele. Nada impede, porém, a alteração manual dessa configuração para outros tipos de bancos.

Nesse arquivo, são configuradas três conexões diferentes: *development*, *test* e *production*, associados respectivamente aos bancos `agenda_development`, `agenda_test` e `agenda_production`.

O banco *agenda_development* é usado na fase de desenvolvimento da aplicação e *agenda_production* em produção. A configuração para *agenda_test* se refere ao banco utilizado para os testes que serão executados, e deve ser mantido separado dos outros pois o framework apagará dados e tabelas constantemente.

Apesar de já estarem configuradas as conexões, o gerador da aplicação não cria os bancos de dados automaticamente. Existe um comando para isso:

```
rake db:create
```

A geração não é automática pois em alguns casos é possível que algum detalhe ou outro de configuração no arquivo **config/database.yml** seja necessário.

6.7 Exercícios - Criando o banco de dados

1) • Pelo terminal entre no diretório do projeto.

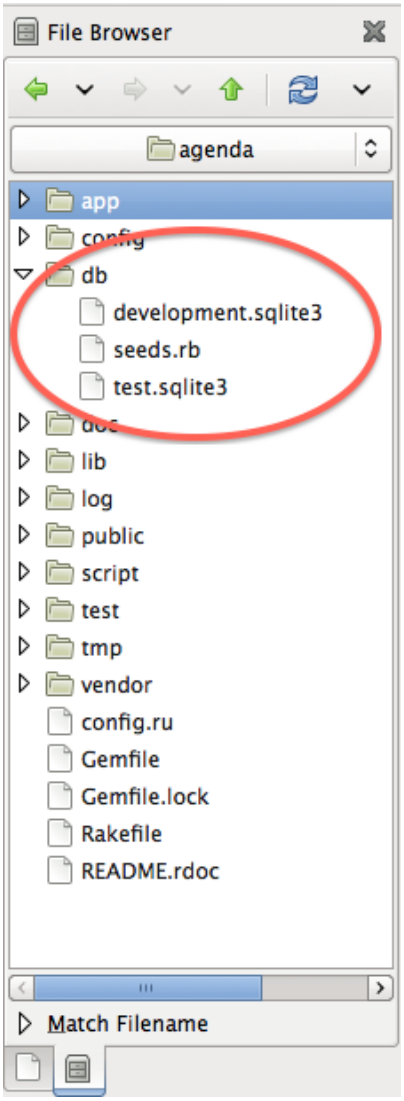
```
cd agenda
```

- Execute o script para criação dos bancos de dados.

```
rake db:create
```

Esse comando não deve exibir nenhuma saída no console, não se preocupe. Porém, se alguma mensagem for exibida, verifique se o comando foi digitado exatamente como demonstrado acima.

2) Verifique que as bases de desenvolvimento e teste foram criadas. Pelo explorador de arquivos do GEdit, verifique a pasta **db** da nossa aplicação. O *Rails* cria arquivos com a extensão **.sqlite3** para serem utilizados como base de dados pelo SQLite3, isso permite que você copie o banco de dados para outra máquina sem a necessidade de processos complexos de configuração e importação/exportação de dados.



6.8 A base da construção: Scaffold

Pensando no propósito de facilitar a execução de tarefas repetitivas, o Rails traz um comando que é muito útil nos casos em que precisamos de um cadastro básico de alguma entidade em nossa aplicação -- um CRUD -- que se chama **scaffold**. *Scaffold* pode ser traduzido para o português como "andaime", ou seja, é uma base, um apoio que facilita a construção de nossa aplicação.

Com esse comando é gerado todo o código necessário para que possamos **listar todos os registros**, **exibir um registro**, **criar um novo registro**, **atualizar um registro** e **excluir um registro** de determinada entidade em nossa aplicação.

6.9 Exercícios - Scaffold

1) Execute o scaffold do seu modelo "Evento":

   a) No terminal, na pasta da aplicação, execute o comando:

   ```
   rails generate scaffold Evento nome:string local:string inicio:datetime
   termino:datetime
   ```

   Veja os diversos arquivos que são criados.

   b) Note que também são gerados arquivos para criação da tabela no banco de dados, onde serão armazenados nossos "eventos", e um arquivo chamado **routes.rb** foi modificado.

   c) (opcional) Criamos atributos dos tipos *string*, *datetime*. Quais outros tipos são suportados? Abra a documentação do Rails (http://api.rubyonrails.org) e procure pela classe **ActiveRecord*ConnectionAdapters*Table**.

6.10 Gerar as tabelas

Juntamente com nossos arquivos foi gerado um arquivo de migração, dentro da pasta **db/migrate**. No cenário mais comum, para cada entidade que criamos é necessário que exista uma tabela no banco de dados, para garantir esse padrão o Rails gera automaticamente um script em Ruby para que as informações no banco de dados sejam consistentes com o que declaramos na aplicação.

Quando executamos o script de *scaffold*, implicitamente declaramos que nossa aplicação tem uma entidade **Evento**, então o Rails gerou o arquivo **db/migrate/<timestamp>_create_eventos.rb** (`<timestamp>` corresponde ao momento da criação do arquivo) com a definição da nossa tabela `eventos` e dos campos necessários.

Agora precisamos executar o script de geração das tabelas. Assim como temos um comando *rake* que cria o banco de dados, também temos um comando para criar as tabelas necessárias para nossa aplicação:

```
rake db:migrate
```

O Rails gera para todas as tabelas por padrão o campo `id` como chave primária. Toda a lógica de utilização dessa chave primária é tratada automaticamente para que o desenvolvedor possa se preocupar cada vez menos com o banco de dados e mais com os modelos.

Versão do banco de dados

Ao executarmos a migração do banco de dados pela primeira vez, será criada uma outra tabela chamada **schema_migrations**. Essa tabela contém uma única coluna (`version`), que indica quais foram as migrações executadas até agora, utilizando para isso o timestamp da última migration executada.

> **Database evolution**
>
> O Rails adota uma estratégia de evolução para o banco de dados, dessa maneira é possível a aplicação evolua gradativamente e o esquema do banco de dados seja incrementado com tabelas e campos em tabelas já existentes conforme o necessário.
>
> Essa característica vai de encontro com as necessidades das metodologias ágeis de desenvolvimento de Software.

6.11 Exercícios - Migrar tabela

1) Vamos executar o script de migração de banco de dados para que a tabela "eventos" seja criada:

   a) Execute o comando pelo terminal, na pasta da nossa aplicação:

   ```
   rake db:migrate
   ```

   b) O Rails inclui uma maneira facilitada de conectarmos ao console interativo do banco de dados para que possamos inspecionar as tabelas e registros:

   ```
   rails dbconsole
   ```

   c) O comando anterior inicia o console interativo, portanto agora será necessário utilizar a sintaxe do SQLite3 para verificar nossas tabelas. Vamos primeiro listar todas as tabelas do nosso banco de dados de desenvolvimento:

   ```
   sqlite> .tables
   ```

   d) Agora vamos solicitar mais informações sobre a tabela *eventos*:

   ```
   sqlite> PRAGMA table_info(eventos);
   ```

   e) Para finalizar a sessão de console de banco de dados, execute o comando a seguir:

   ```
   sqlite> .quit
   ```

A sintaxe do console interativo do banco de dados depende do tipo de banco utilizado. Se você escolher, por exemplo, utilizar o MySQL os comandos para listar as tabelas do banco de dados é `show tables;`, e para obter detalhes sobre a tabela é `describe eventos;`.

6.12 Server

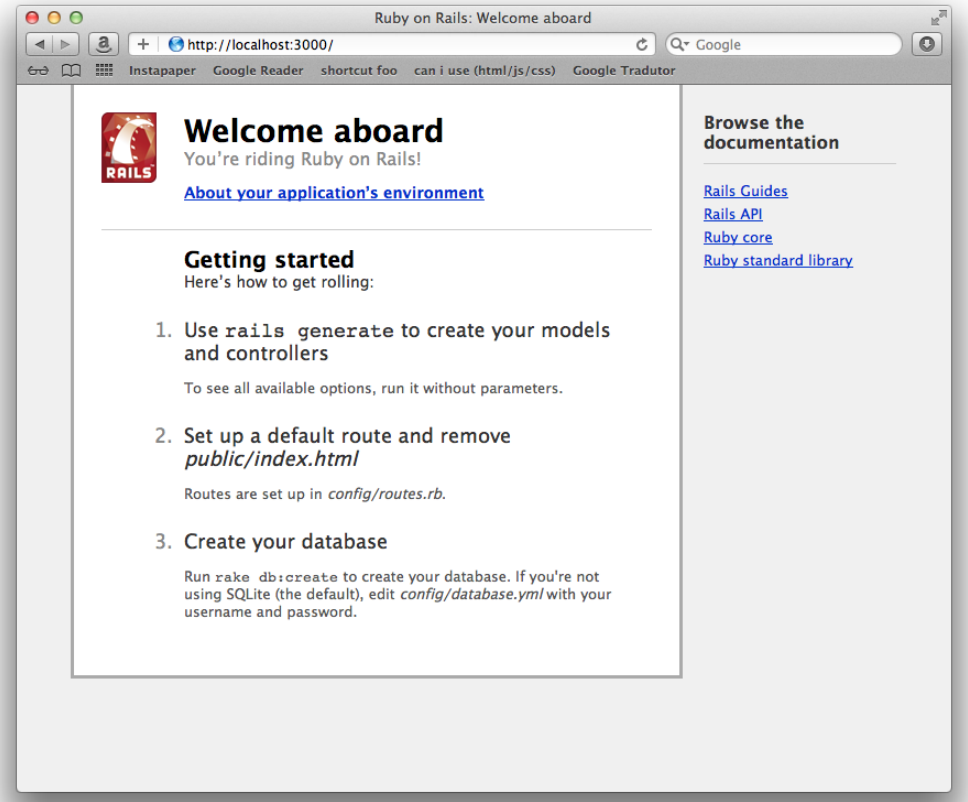
Agora que já geramos o código necessário em nossa aplicação, criamos o banco de dados e as tabelas necessárias, precisamos colocar nossa aplicação em um servidor de aplicações Web que suporte aplicações feitas com o Ruby on Rails. Só assim os usuários poderão acessar nossa aplicação.

O próprio conjunto de ferramentas embutidas em uma instalação padrão de um ambiente Rails já inclui um servidor simples, suficiente para que o desenvolvedor possa acessar sua aplicação através do *browser*. Esse servidor, o **WEBrick**, não é indicado para aplicações em produção por ser muito simples e não contar com recursos mais avançados necessários para colocar uma aplicação "no ar" para um grande número de usuários.

Podemos iniciar o servidor através do comando *rails server* no terminal:

```
rails server
```

Após o processo de inicialização, podemos acessar nossa aplicação no navegador. Por padrão o servidor disponibiliza nossa aplicação na porta 3000, podemos visitar nossa aplicação em http://localhost:3000.



Quando geramos o *scaffold* de eventos, foram geradas telas simples para listar todos registros de eventos, exibir um evento e os formulários para criar um novo evento e editar um evento existente. As telas são simples, mas são totalmente funcionais.

Ao visitar "http://localhost:3000/eventos", você é redirecionado para a listagem de eventos.

6.13 Exercícios - Iniciando o servidor

1) a) Inicie o servidor com o comando:

```
rails server
```

b) Acesse a aplicação pela url: http://localhost:3000/eventos

6.14 Documentação do Rails

O Ruby tem uma ferramenta capaz de gerar documentação do nosso código a partir dos comentários que podemos fazer dentro de uma classe ou imediatamente antes das declarações de métodos, essa ferramenta é o RDoc.

A documentação da versão atual do Rails está disponível em **http://api.rubyonrails.org**.

Há diversos outros sites que fornecem outras versões da documentação, ou modos alternativos de exibição e organização.

- http://www.railsapi.com
- http://www.gotapi.com
- http://apidock.com/rails

O RubyGems também oferece a documentação para cada uma das gems que estão instaladas, para isso basta iniciar o servidor de documentação embutido:

```
gem server
```

Este comando inicia um servidor WebRick na porta 8808. Basta acessar pelo browser para ver o RDoc de todos os gems instalados.

A última alternativa é gerar a documentação através do rake:

```
rails docs
rake doc:rails
```

Depois de executar a task `doc:rails`, a documentação estará disponível no diretório *docs/api/*. A documentação do Ruby (bem como a biblioteca padrão) pode ser encontrada em **http://ruby-doc.org**.

Existem ainda excelentes guias e tutoriais oficiais disponíveis. É **obrigatório** para todo desenvolvedor Rails passar por estes guias: **http://guides.rubyonrails.org**.

Além desses guias, existe um site que lista as principais gems que podem ser usadas para cada tarefa: **http://ruby-toolbox.com/**.

A comunidade Ruby e Rails também costuma publicar excelentes *screencasts* (vídeos) com aulas e/ou palestras sobre diversos assuntos relacionados a Ruby e Rails. A lista está sempre crescendo. Aqui estão alguns dos principais:

- **http://railscasts.com** - Vídeos pequenos abordando algumas práticas e ferramentas interessantes, mantidos por Ryan Bates. Alguns vídeos são gratuitos.
- **http://peepcode.com** - Verdadeiras vídeo-aulas, algumas chegam a 2h de duração. São pagos, cerca de US$ 9,00 por vídeo.
- **http://www.confreaks.com** - Famosos por gravar diversas conferências sobre Ruby e Rails.

6.15 Exercício opcional - Utilizando a documentação

No formulário de cadastro de eventos podemos verificar que, apesar de já criar os formulários com os componentes adequados para que o usuário possa escolher uma data correta, a lista de anos disponíveis exibe somente alguns anos recentes.

Um dos requisitos possíveis para nossa aplicação é que o ano inicial do evento seja no mínimo o ano atual, para evitar que algum usuário cadastre um evento com uma data de fim no passado.

Para fazer essa alteração na aplicação será preciso editar o código gerado pelo *Rails* responsável por exibir a lista de anos, comece a listar a partir do ano atual:

1) Verifique no arquivo *app/views/eventos/_form.html.erb* um método chamado datetime_select:

   ```
   <%= f.datetime_select :termino %>
   ```

   Encontre a documentação desse método (em **ActionView*Helpers*DateHelper**). Como podemos indicar que queremos exibir anos a partir do ano atual?

   ```
   <%= f.datetime_select :termino, start_year: 2012 %>
   ```

Capítulo 7

Active Record

"Não se deseja aquilo que não se conhece"
– Ovídio

Nesse capítulo começaremos a desenvolver um sistema utilizando Ruby on Rails com recursos mais avançados.

7.1 Motivação

Queremos criar um sistema de qualificação de restaurantes. Esse sistema terá clientes que qualificam os restaurantes visitados com uma nota, além de informar quanto dinheiro gastaram. Os clientes terão a possibilidade de deixar comentários para as qualificações feitas por eles mesmos ou a restaurantes ainda não visitados. Além disso, os restaurantes terão pratos, e cada prato a sua receita.

O site http://www.tripadvisor.com possui um sistema similar para viagens, onde cada cliente coloca comentários sobre hotéis e suas visitas feitas no mundo inteiro.

7.2 Exercícios: Controle de Restaurantes

1) Crie um novo projeto chamado `vota_prato`:

   a) No terminal, garanta que não está no diretoria do projeto anterior.

   b) Digite o comando `rails new vota_prato -d mysql`

   c) Observe o log de criação do projeto:

7.3 Modelo - O "M" do MVC

Models são os modelos que serão usados nos sistemas: são as entidades que serão armazenadas em um banco. No nosso sistema teremos modelos para representar um **Cliente**, um **Restaurante** e uma **Qualificação**, por exemplo.

O componente de Modelo do Rails é um conjunto de classes que usam o `ActiveRecord`, uma classe ORM que mapeia objetos em tabelas do banco de dados. O `ActiveRecord` usa convenções de nome para determinar os mapeamentos, utilizando uma série de regras que devem ser seguidas para que a configuração seja a mínima possível.

ORM

ORM (*Object-Relational Mapping*) é um conjunto de técnicas para a transformação entre os modelos orientado a objetos e relacional.

7.4 ActiveRecord

É um framework que implementa o acesso ao banco de dados de forma transparente ao usuário, funcionando como um Wrapper para seu modelo. Utilizando o conceito de *Conventions over Configuration*, o ActiveRecord adiciona aos seus modelos as funções necessárias para acessar o banco.

`ActiveRecord::Base` é a classe que você deve estender para associar seu modelo com a tabela no Banco de Dados.

7.5 Rake

Rake é uma ferramenta de build, escrita em Ruby, e semelhante ao **make** e ao **ant**, em escopo e propósito.

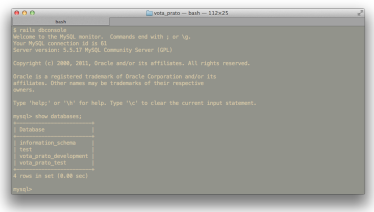
Rake tem as seguintes funcionalidades:

- Rakefiles (versão do rake para os Makefiles) são completamente definidas em sintaxe Ruby. Não existem arquivos XML para editar, nem sintaxe rebuscada como a do Makefile para se preocupar.
- É possível especificar tarefas com pré-requisitos.
- Listas de arquivos flexíveis que agem como arrays, mas sabem como manipular nomes de arquivos e caminhos (paths).
- Uma biblioteca de tarefas pré-compactadas para construir rakefiles mais facilmente.

Para criar nossas bases de dados, podemos utilizar a rake task **db:create**. Para isso, vá ao terminal e dentro do diretório do projeto digite:

```
$ rake db:create
```

O *Rails* criará dois databases no mysql: `vota_prato_development`, `vota_prato_test`.

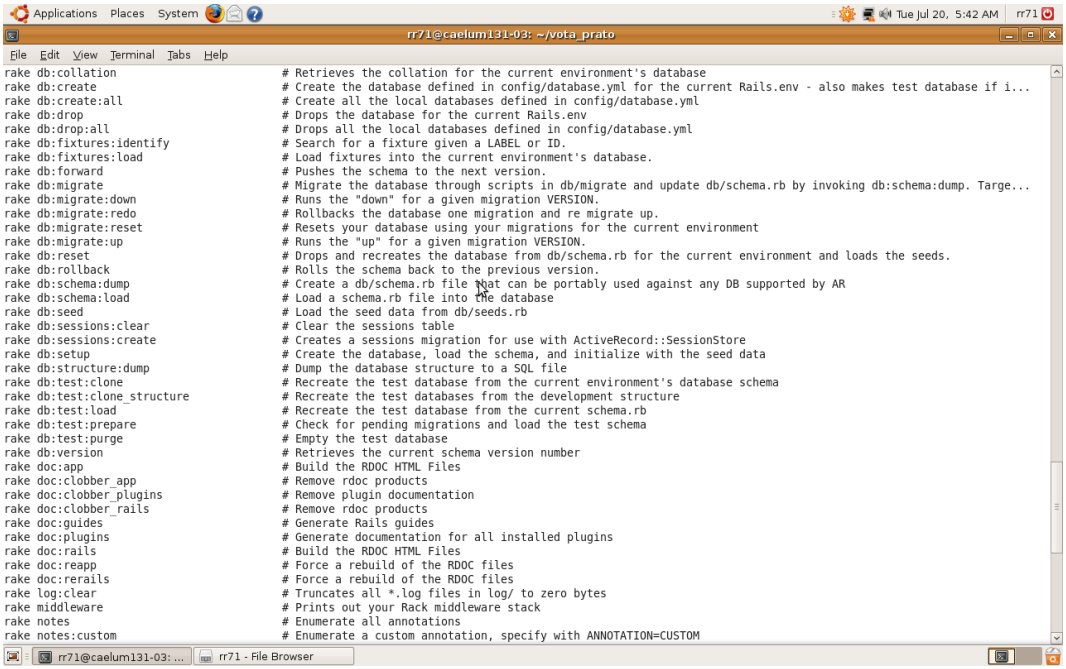


RAKE DB:CREATE:ALL

Uma outra tarefa disponível em uma aplicação *Rails* e a `rake db:create:all`. Além de criar os dois databases já citados, ela também é responsável por criar o database `vota_prato_production`. No ambiente de desenvolvimento, é bem comum trabalharmos apenas com os databases de teste e desenvolvimento.

Para ver todas as tasks rake disponíveis no seu projeto podemos usar o comando (na raiz do seu projeto):

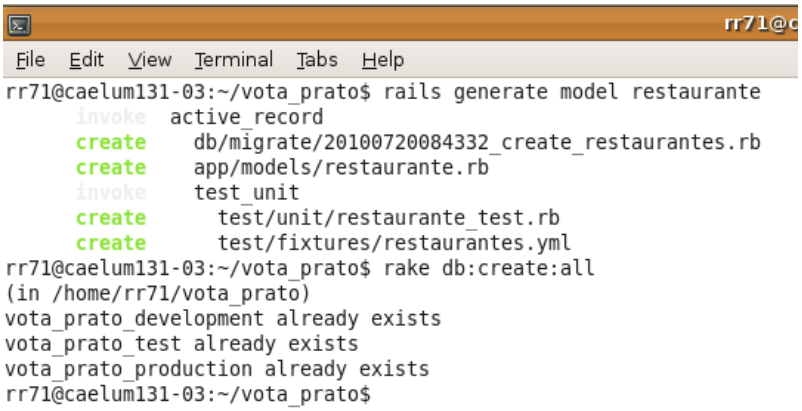
```
$ rake -T
```



7.6 Criando Modelos

Agora vamos criar o modelo do Restaurante. Para isso, temos um gerador específico para **model** através do comando *rails generate model restaurante*.

Repare que o Rails gerou uma série de arquivos para nós.



7.7 Migrations

Migrations ajudam a gerenciar a evolução de um esquema utilizado por diversos bancos de dados. Foi a solução encontrada para o problema de como adicionar uma coluna no banco de dados local e propagar essa mudança para os demais desenvolvedores de um projeto e para o servidor de produção.

Com as migrations, podemos descrever essas transformações em classes que podem ser controladas por sistemas de controle de versão (por exemplo, git) e executá-las em diversos bancos de dados.

Sempre que executarmos a tarefa *Generator -> model*, o Rails se encarrega de criar uma migration inicial, localizado em **db/migrate**.

`ActiveRecord::Migration` é a classe que você deve estender ao criar uma migration.

Quando geramos nosso modelo na seção anterior, Rails gerou para nós uma migration (**db/migrate/<timestamp>_create_restaurantes.rb**). Vamos agora editar nossa migration com as informações que queremos no banco de dados.

Queremos que nosso restaurante tenha um nome e um endereço. Para isso, devemos acrescentar as chamadas de método abaixo:

```
t.string :nome, limit: 80
t.string :endereco
```

Faça isso dentro do método `change` da classe `CreateRestaurantes`. Sua migration deverá ficar como a seguir:

```
class CreateRestaurantes < ActiveRecord::Migration
  def change
    create_table :restaurantes do |t|
      t.string :nome, limit: 80
      t.string :endereco
      t.timestamps
    end
```

```
  end
end
```

Supondo que agora lembramos de adicionar a especialidade do restaurante. Como fazer? Basta usar o outro gerador (*Generator*) do rails que cria migration. Por exemplo:

```
$ rails generate migration add_column_especialidade_to_restaurante especialidade
```

Um novo arquivo chamado `<timestamp>_add_column_especialidade_to_restaurante.rb` será criado pelo *Rails*, e o código da migração gerada será como a seguir:

```
class AddColumnEspecialidadeToRestaurante < ActiveRecord::Migration
  def change
    add_column :restaurantes, :especialidade, :string
  end
end
```

Note que através do nome da migração e do parâmetro que passamos no gerador, o nome da coluna que queremos adicionar, nesse caso **especialidade**, o *Rails* deduziu que a migração é para adicionar a coluna e já fez o trabalho braçal para você!

Vamos apenas adicionar mais alguns detalhes a essa migração, vamos limitar o tamanho da string que será armazenada nesse campo para 40 caracteres:

```
class AddColumnEspecialidadeToRestaurante < ActiveRecord::Migration
  def change
    add_column :restaurantes, :especialidade, :string, limit: 40
  end
end
```

7.8 Exercícios: Criando os modelos

1) Crie nosso banco de dados

   a) Entre no terminal, no diretório do projeto

   b) execute o comando:

   ```
   $ rake db:create
   ```

   No terminal, acesse o mysql e verifique que os databases foram criados:

   ```
   $ rails dbconsole
   mysql> show databases;
   mysql> quit
   ```

2) Crie o modelo do Restaurante

   a) Novamente no Terminal, no diretório do projeto

   b) execute **rails generate model restaurante**

3) Edite seu script de migração do modelo "restaurante" para criar os campos nome e endereço:

   a) Abra o arquivo `db/migrate/<timestamp>_create_restaurantes.rb`

   b) Adicione as linhas:

```
t.string :nome, limit: 80
t.string :endereco
```

   O código final de sua migration deverá ser como o que segue:

```
class CreateRestaurantes < ActiveRecord::Migration
  def change
    create_table :restaurantes do |t|
      t.string :nome, limit: 80
      t.string :endereco
      t.timestamps
    end
  end
end
```

4) Migre as tabelas para o banco de dados:

   a) Vá ao Terminal

   b) Execute a tarefa db:migrate:

```
$ rake db:migrate
```

   Verifique no banco de dados se as tabelas foram criadas:

```
$ rails dbconsole
mysql> use vota_prato_development;
mysql> desc restaurantes;
mysql> quit
```

5) Adicione a coluna especialidade ao nosso modelo "restaurante":

   a) Vá novamente ao Terminal e digite:

```
$ rails generate migration add_column_especialidade_to_restaurante especialidade
```

   b) Abra o arquivo `db/migrate/<timestamp>_add_column_especialidade_to_restaurante.rb`

   c) Altere o limite de caracteres da coluna especialidade como a seguir:

```
add_column :restaurantes, :especialidade, :string, limit: 40
```

Seu arquivo ficará assim:

```
class AddColumnEspecialidadeToRestaurante < ActiveRecord::Migration
  def change
    add_column :restaurantes, :especialidade, :string, limit: 40
  end
end
```

d) Para efetivar a mudança no banco de dados execute a seguinte tarefa:

```
rake db:migrate
```

A saída do comando será semelhante a seguinte:

```
==  CreateRestaurantes: migrating ============================================
-- create_table(:restaurantes)
   -> 0.7023s
==  CreateRestaurantes: migrated (0.7025s) ===================================

==  AddColumnEspecialidadeToRestaurante: migrating ===========================
-- add_column(:restaurantes, :especialidade, :string, {:limit=>40})
   -> 0.9402s
==  AddColumnEspecialidadeToRestaurante: migrated (0.9404s) ==================
```

e) Olhe novamente no banco de dados, veja que as tabelas foram realmente criadas.

```
$ rails dbconsole
mysql> use vota_prato_development;
mysql> show tables;
```

E o resultado será semelhante ao seguinte:

```
| Tables_in_vota_prato_development |

| restaurantes                     |
| schema_migrations                |

2 rows in set (0.00 sec)
```

6) (Opcional) Utilizamos o método **add_column** na nossa *migration* para adicionar uma nova coluna. O que mais poderia ser feito? Abra a documentação e procure pelo módulo **ActiveRecord*ConnectionAdapters*SchemaStatements**.

7.9 Manipulando nossos modelos pelo console

Podemos utilizar o console para escrever comandos Ruby, e testar nosso modelo. A grande vantagem disso, é que não precisamos de controladores ou de uma view para testar se nosso modelo funciona de acordo com

o esperado e se nossas regras de validação estão funcionando. Outra grande vantagem está no fato de que se precisarmos manipular nosso banco de dados, ao invés de termos de conhecer a sintaxe sql e digitar a query manualmente, podemos utilizar código ruby e manipular através do nosso console.

Para criar um novo restaurante, podemos utilizar qualquer um dos jeitos abaixo:

```
r = Restaurante.new
r.nome = "Fasano"
r.endereco = "Av. dos Restaurantes, 126"
r.especialidade = "Comida Italiana"
r.save

r = Restaurante.new do |r|
  r.nome = "Fasano"
  r.endereco = "Av. dos Restaurantes, 126"
  r.especialidade = "Comida Italiana"
end
r.save
```

Uma outra forma possível para a criação de objetos do Active Record é usando um *hash* como parâmetro no construtor. As chaves do hash precisam coincidir com nomes das propriedades as quais queremos atribuir os valores. Veja o exemplo a seguir:

```
r = Restaurante.new nome: "Fasano",
                    endereco: "Av. dos Restaurantes, 126",
                    especialidade: "Comida Italiana"
r.save
```

Porém o Active Record tem uma restrição quanto ao uso dessa funcionalidade. É preciso que você especifique exatamente quais são as propriedades que podem ter seu valor atribuído a partir do *hash*.

É preciso dizer isso explicitamente através da invocação do método `attr_accessible`. Vamos editar a classe `Restaurante` para que o código fique como o seguinte:

```
class Restaurante < ActiveRecord::Base
  attr_accessible :nome, :endereco, :especialidade
end
```

Pronto! Agora você já pode criar objetos com as propriedades populadas através de um *hash*. Além disso, também pode invocar um método de classe em `Restaurante` que cria um objeto e já armazena os dados no banco, sem precisar da invocação ao método save, veja:

```
Restaurante.create nome: "Fasano",
                   endereco: "Av. dos Restaurantes, 126",
                   especialidade: "Comida Italiana"
```

MASS ASSIGNMENT

Essa ideia de precisar especificar quais campos podem ser atribuídos através de um *hash* é referida por muitos como "mass assignment whitelist". A ideia é que o programador deve decidir quais propriedades de um objeto podem ser atribuídas diretamente.

Em versões anteriores do framework, toda propriedade era passível de atribuição, a não ser que o programador se lembrasse de adicionar uma invocação ao método `attr_accessible`. O problema disso é que, se por algum descuido, o programador esquecer de estabelecer exatamente quais propriedades estão abertas a atribuição, algumas falhas de segurança podem ocorrer.

Atualmente o padrão é justamente o inverso. Nenhuma propriedade pode ser atribuída diretamente a partir dos valores de um *hash*, caso queira isso, é preciso especificar através da invocação do *attr_accessible*.

Você pode ler mais a respeito nesse post: http://blog.caelum.com.br/seguranca-de-sua-aplicacao-e-os-frameworks-ataque-ao-github/

Note que o comando save efetua a seguinte ação: se o registro não existe no banco de dados, cria um novo registro; se já existe, atualiza o registro existente.

Existe também o comando save!, que tenta salvar o registro, mas ao invés de apenas retornar **"false"** se não conseguir, lança a exceção `RecordNotSaved`.

Para atualizar um registro diretamente no banco de dados, podemos fazer:

```
Restaurante.update(1, {nome: "1900"})
```

Para atualizar múltiplos registros no banco de dados:

```
Restaurante.update_all("especialidade = 'Massas'")
```

Ou, dado um objeto r do tipo Restaurante, podemos utilizar:

```
r.update_attribute(:nome, "1900")
```

```
r.update_attributes nome: "1900", especialidade: "Pizzas"
```

Existe ainda o comando update_attributes!, que chama o comando save! ao invés do comando save na hora de salvar a alteração.

Para remover um restaurante, também existem algumas opções. Todo `ActiveRecord` possui o método `destroy`:

```
restaurante = Restaurante.first
restaurante.destroy
```

Para remover o restaurante de id **1**:

```
Restaurante.destroy(1)
```

Para remover os restaurantes de ids **1, 2 e 3**:

```
restaurantes = [1,2,3]
Restaurante.destroy(restaurantes)
```

Para remover **todos** os restaurantes:

```
Restaurante.destroy_all
```

Podemos ainda remover todos os restaurantes que obedeçam determinada condição, por exemplo:

```
Restaurante.destroy_all(especialidade: "italiana")
```

Os métodos destroy sempre fazem primeiro o `find(id)` para depois fazer o `destroy()`. Se for necessário evitar o `SELECT` antes do `DELETE`, podemos usar o método `delete()`:

```
Restaurante.delete(1)
```

7.10 Exercícios: Manipulando registros

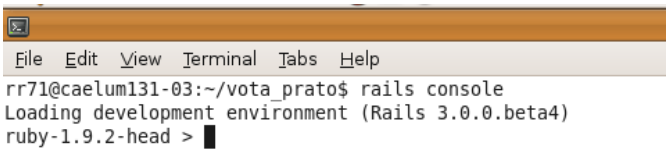
Teste a manipulação de registros pelo console.

1) Edite o arquivo localizado em `app/models/restaurante.rb`. Na declaração da classe invoque o método `attr_accessible` como no exemplo abaixo:

```
class Restaurante < ActiveRecord::Base
  attr_accessible :nome, :endereco, :especialidade
end
```

2) Insira um novo restaurante

   a) Para ter acesso ao Console, basta digitar **rails console** no Terminal



b) Digite:

```
r = Restaurante.new nome: "Fasano",
                    endereco: "Av. dos Restaurantes, 126",
                    especialidade: "Comida Italiana"
```

c) Olhe seu banco de dados:

```
$ rails dbconsole
mysql> select * from restaurantes;
```

d) Volte para o Console e digite:

```
r.save
```

e) Olhe seu banco de dados novamente:

```
$ rails dbconsole
mysql> select * from restaurantes;
```

3) Atualize seu restaurante

a) Digite:

```
r.update_attributes nome: "1900"
```

b) Olhe seu banco de dados novamente:

```
$ rails dbconsole
mysql> select * from restaurantes;
```

4) Vamos remover o restaurante criado:

a) Digite

```
Restaurante.destroy(1)
```

b) Olhe seu banco de dados e veja que o restaurante foi removido

```
$ rails dbconsole
mysql> select * from restaurantes;
```

7.11 Exercícios Opcionais

1) Teste outras maneiras de efetuar as operações do exercício anterior.

7.12 Finders

O ActiveRecord possui o método **"find"** para realizar buscas. Esse método, aceita os seguintes parâmetros:

```
Restaurante.all   # retorna todos os registros
Restaurante.first # retorna o primeiro registro
Restaurante.last  # retorna o último registro
```

Ainda podemos passar para o método find uma lista com os id's dos registros que desejamos:

```
r = Restaurante.find(1)
varios = Restaurante.find(1,2,3)
```

Além desses, podemos definir condições para nossa busca (como o `SELECT` do MySQL). Existem diversas formas de declararmos essas condições:

```
Restaurante.where("nome = 'Fasano' and especialidade = 'massa'")
Restaurante.where(["nome = ? and especialidade = ?",
    'Fasano', 'massa'])
Restaurante.where(["nome = :nome and especialidade = :especialidade", {
      nome: "Fasano", especialidade: "Massa"}])
Restaurante.where({nome: "Fasano", especialidade: "massa" })
```

Essas quatro formas fazem a mesma coisa. Procuram por registros com o campo `nome = "Fasano"` e o campo `especialidade = "massa"`.

Existem ainda os chamados dynamic finders:

```
Restaurante.where(["nome = ? AND especialidade = ?",
    "Fasano", "italiana"])
```

poderia ser escrito como:

```
find_all_by_nome_and_especialidade("Fasano", "italiana")
```

Temos ainda o **"find_or_create_by"**, que retorna um objeto se ele existir, caso contrário, cria o objeto no banco de dados e retorna-o:

```
Restaurante.find_or_create_by_nome("Fasano")
```

Para finalizar, podemos chamar outros métodos encadeados para fazer querys mais complexas:

- .order - define a ordenação. Ex: "created_at DESC, nome".
- .group - nome do atributo pelo qual os resultados serão agrupados. Efeito idêntico ao do comando SQL GROUP BY.
- .limit - determina o limite do número de registros que devem ser retornados
- .offset - determina o ponto de início da busca. Ex: para offset = 5, iria pular os registros de 0 a 4.
- .include - permite carregar relacionamentos na mesma consulta usando `LEFT OUTER JOINS`.

Exemplo mais completo:

```
Restaurante.where('nome like :nome', {nome: '%teste%'}).
      order('nome DESC').limit(20)
```

> **Para Saber Mais - Outras opções para os finders**
>
> Existem mais opções, como o **":lock"**, que podem ser utilizadas, mas não serão abordadas nesse curso. Você pode consultá-las na documentação da API do Ruby on Rails.

7.13 Exercícios: Buscas dinâmicas

1) Vamos testar os métodos de busca:
   a) Abra o console (`rails console` no Terminal)
   b) Digite:

      ```
      Restaurante.first
      ```

   c) Aperte **enter**
   d) Digite:

      ```
      Restaurante.all
      ```

   e) Aperte **enter**
   f) Digite:

      ```
      Restaurante.find(1)
      ```

   g) Aperte **enter**

h) Digite:

```
Restaurante.where(["nome = ? AND especialidade = ?",
                                    "Fasano", "Comida Italiana"])
```

i) Aperte **enter**

j) Digite:

```
Restaurante.find_all_by_nome_and_especialidade("Fasano",
                                   "Comida Italiana")
```

k) Aperte **enter**

l) Digite:

```
Restaurante.order("especialidade DESC").limit(1)
```

m) Aperte **enter**

7.14 Validações

Ao inserir um registro no banco de dados é bem comum a entrada de dados inválidos.

Existem alguns campos de preenchimento obrigatório, outros que só aceitem números, que não podem conter dados já existentes, tamanho máximo e mínimo etc.

Para ter certeza que um campo foi preenchido antes de salvar no banco de dados, é necessário pensar em três coisas: "como validar a entrada?", "qual o campo a ser validado?" e "o que acontece ao tentar salvar uma entrada inválida?".

Para validar esses registros, podemos implementar o método `validate` em qualquer `ActiveRecord`, porém o Rails disponibiliza alguns comandos prontos para as validações mais comuns. São eles:

- validates_presence_of: verifica se um campo está preenchido;
- validates_size_of: verifica o comprimento do texto do campo;
- validates_uniqueness_of: verifica se não existe outro registro no banco de dados que tenha a mesma informação num determinado campo;
- validates_numericality_of: verifica se o preenchimento do campo é numérico;
- validates_associated: verifica se o relacionamento foi feito corretamente;
- etc...

Todos estes métodos disponibilizam uma opção (**:message**) para personalizar a mensagem de erro que será exibida caso a regra não seja cumprida. Caso essa opção não seja utilizada, será exibida uma mensagem padrão.

Toda mensagem de erro é gravada num hash chamado errors, presente em todo ActiveRecord.

Além dos validadores disponibilizados pelo rails, podemos utilizar um validador próprio:

```
validate :garante_alguma_coisa

def garante_alguma_coisa
  errors.add_to_base("Deve respeitar nossa regra") unless campo_valido?
end
```

Repare que aqui, temos que incluir manualmente a mensagem de erro padrão do nosso validador.

Se quisermos que o nome do nosso restaurante comece com letra maiúscula, poderíamos fazer:

```
validate :primeira_letra_deve_ser_maiuscula

private
def primeira_letra_deve_ser_maiuscula
  errors.add("nome",
      "primeira letra deve ser maiúscula") unless nome =~ /[A-Z].*/
end
```

Além dessas alternativas, o Rails 3 trouxe uma nova maneira de criar validações que facilita os casos onde temos vários validadores para o mesmo campo, como por exemplo:

```
validates :nome, presence: true, uniqueness: true, length: {maximum: 50}
```

equivalente a:

```
validates_presence_of :nome
validates_uniqueness_of :nome
validates_length_of :nome, :maximum => 50
```

Modificadores de acesso

Utilizamos aqui, o modificador de acesso **private**. A partir do ponto que ele é declarado, todos os métodos daquela classe serão privados, a menos que tenha um outro modificador de acesso que modifique o acesso a outros métodos.

Validadores prontos

Esse exemplo poderia ter sido reescrito utilizando o validador **"validates_format_of"**, que verifica se um atributo confere com uma determinada expressão regular.

Encoding de arquivos no Ruby e as classes de Model

Devido ao encoding dos arquivos no Ruby 1.9 ser como padrão ASCII-8BIT, ao utilizarmos algum carácter acentuado no código fonte, é necessário indicarmos um encoding que suporte essa acentuação. Geralmente, esse encoding será o UTF-8. Portanto, temos que adicionar no começo do arquivo fonte a linha:

```
#encoding: utf-8
```

7.15 Exercícios: Validações

1) Para nosso restaurante implementaremos a validação para que o campo nome, endereço e especialidade não possam ficar vazios, nem que o sistema aceite dois restaurantes com o mesmo nome e endereço.

   a) Abra o modelo do restaurante (**app/models/restaurante.rb**)

   b) inclua as validações:

```
validates_presence_of :nome, message: "deve ser preenchido"
validates_presence_of :endereco, message: "deve ser preenchido"
validates_presence_of :especialidade, message: "deve ser preenchido"

validates_uniqueness_of :nome, message: "nome já cadastrado"
validates_uniqueness_of :endereco, message: "endereço já cadastrado"
```

   c) Por utilizarmos acentuação em nosso arquivo de modelo, adicione a diretiva (comentário) que indica qual o encoding em que o Ruby deve interpretar **no início do seu arquivo**.

```
#encoding: utf-8
```

2) Inclua a validação da primeira letra maiúscula:

```
validate :primeira_letra_deve_ser_maiuscula

private
def primeira_letra_deve_ser_maiuscula
  errors.add(:nome,
      "primeira letra deve ser maiúscula") unless nome =~ /[A-Z].*/
end
```

3) Agora vamos testar nossos validadores:

   a) Abra o Terminal

b) Entre no Console (rails console)

c) Digite:

```
r = Restaurante.new nome: "fasano",
      endereco: "Av. dos Restaurantes, 126"
r.save
```

d) Verifique a lista de erros, digitando:

```
r.valid?          # verifica se objeto passa nas validações
r.errors.empty?   # retorna true/false indicando se há erros ou não
r.errors.count    # retorna o número de erros
r.errors[:nome]   # retorna apenas o erro do atributo nome

r.errors.each {|field, msg| puts "#{field} - #{msg}"}
```

7.16 Pluralização com inflections

Repare que o Rails pluralizou o termo **restaurante** automaticamente. Por exemplo, o nome da tabela é **restaurantes**. O Rails se baseia nas regras gramaticais da lingua inglesa para fazer essa pluralização automática.

Apesar de **restaurante** ter sido plurificado corretamente, outros termos como **qualificacao** por exemplo, será plurificado para **qualificacaos**, ao invés de **qualificacoes**. Podemos fazer o teste no console do Rails, a String tem os métodos `pluralize` e `singularize` para fazer, respectivamente, pluralização e singularização do termo:

```
"qualificacao".pluralize # => "qualificacaos"
"qualificacoes".singularize # => "qualificacoe"
```

Outro termo onde encontraremos problema é em “receita":

```
"receita".pluralize # => "receita"
"receitas".singularize # => "receita"
```

Apesar da singularização estar correta, a pluralização não ocorre, isso acontece por que de acordo com as regras gramaticais do inglês, a palavra **receita** seria plural de **receitum**. Testando no console obteremos o seguinte resultado:

```
"receitum".pluralize # => "receita"
```

Ou seja, teremos que mudar as regras padrões utilizadas pelo Rails. Para isso, iremos editar o arquivo **config/initializers/inflections.rb**, adicionando uma linha indicando que a pluralização das palavras **qualificacao** e **receita** é, respectivamente, **qualificacoes** e **receitas**:

```
activesupport::inflector.inflections do |inflect|
  inflect.irregular 'qualificacao', 'qualificacoes'
  inflect.irregular 'receita', 'receitas'
end
```

Feito isso, podemos testar nossas novas regras utilizando o console:

```
"qualificacao".pluralize # => "qualificacoes"
"qualificacoes".singularize # => "qualificacao"
"receita".pluralize # => "receitas"
"receitas".singularize # => "receita"
```

7.17 Exercícios - Completando nosso modelo

1) Vamos corrigir a pluralização da palavra 'receita'

   a) Abra o arquivo **"config/initializers/inflections.rb"**

   b) Adicione as seguintes linhas ao final do arquivo:

   ```
   ActiveSupport::Inflector.inflections do |inflect|
     inflect.irregular 'receita', 'receitas'
   end
   ```

   > **BRAZILIAN-RAILS**
   >
   > Existe um plugin chamado **Brazilian Rails** que é um conjunto de gems para serem usadas com Ruby e com o Ruby on Rails e tem como objetivo unir alguns recursos úteis para os desenvolvedores brasileiros.

2) a) Execute o console do Rails:

   ```
   $ rails console
   ```

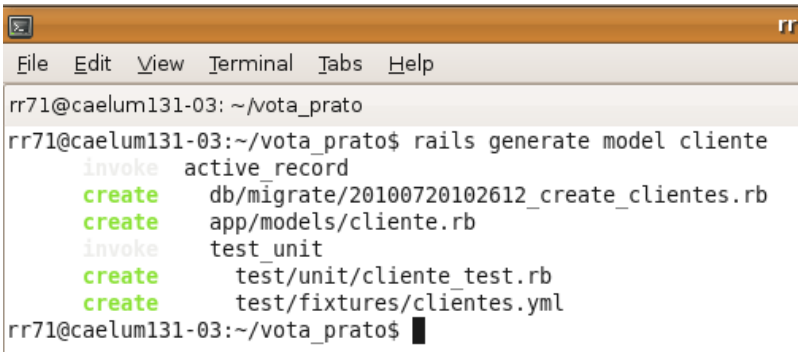
   b) No Console do Rails digite:

   ```
   "receita".pluralize
   ```

3) Vamos criar o nosso modelo Cliente, bem como sua migration:

   a) No terminal, digite:

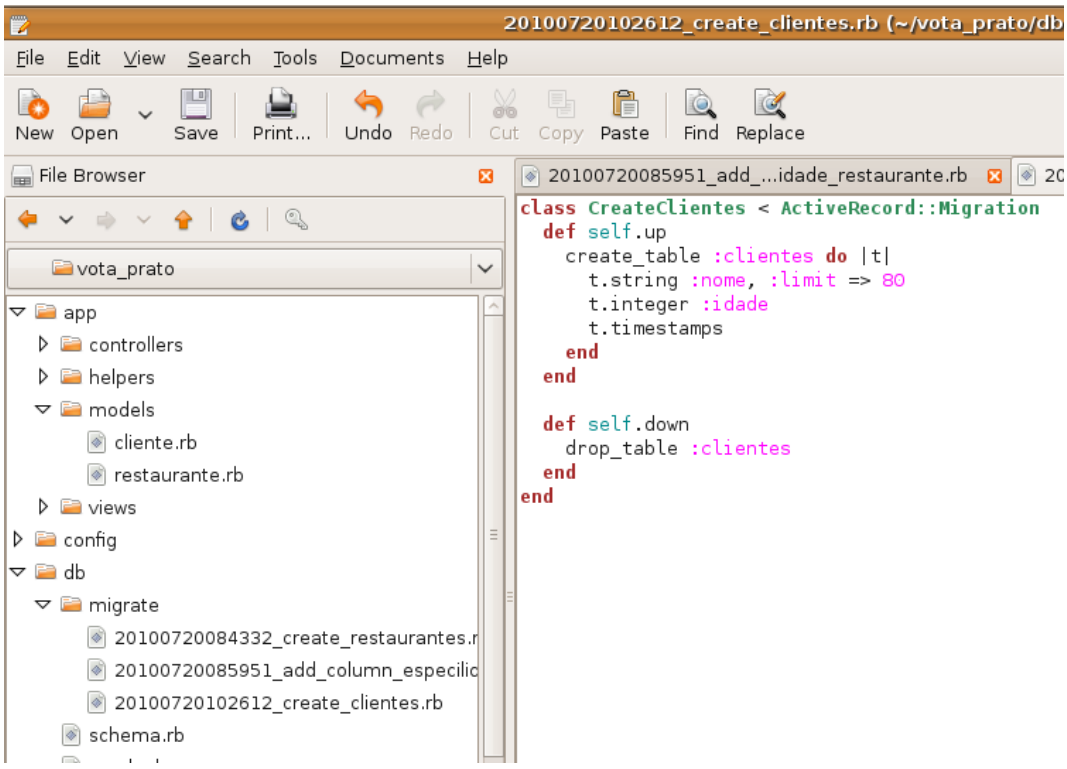
   ```
   $ rails generate model cliente
   ```



b) Abra o arquivo **"db/migrate/<timestamp>_create_clientes.rb"**

c) Edite método change para que ele fique como a seguir:

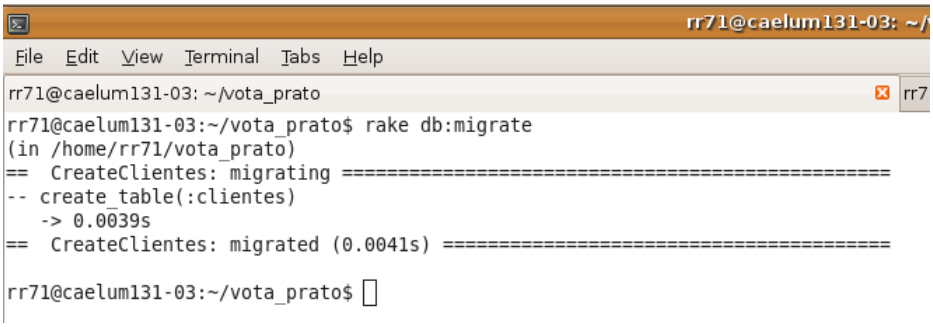
```
create_table :clientes do |t|
  t.string :nome, limit: 80
  t.integer :idade
  t.timestamps
end
```



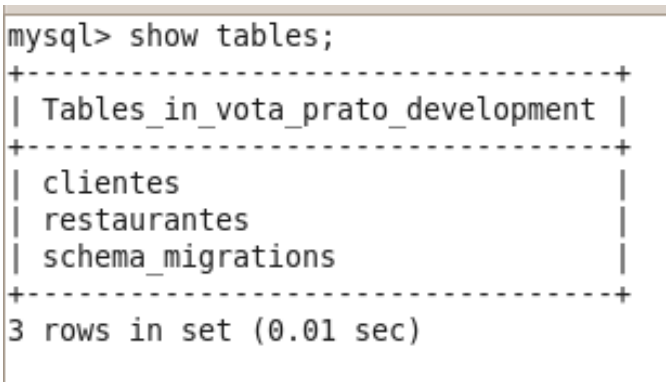
d) Vamos efetivar a migration usando o comando:

```
$ rake db:migrate
```

e) Olhe no console o que foi feito



f) Olhe no banco de dados!



4) Configure as propriedades adequadas para possibilitar o *mass assignment* na classe `Cliente`. Edite a definição criada no arquivo `app/models/cliente.rb` para adicionar a chamada ao `attr_accessible`:

```
class Cliente < ActiveRecord::Base
  attr_accessible :nome, :idade
end
```

5) Vamos agora fazer as validações no modelo **"cliente"**:

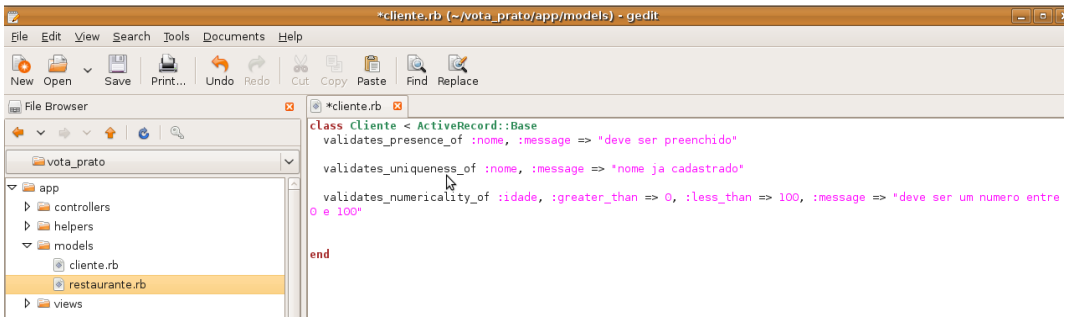
a) Abra o arquivo **"app/models/cliente.rb"**

b) Adicione as seguintes linhas:

```
validates_presence_of :nome, message: " - deve ser preenchido"

validates_uniqueness_of :nome, message: " - nome já cadastrado"

validates_numericality_of :idade, greater_than: 0,
                                  less_than: 100,
                                  message: " - deve ser um número entre 0 e 100"
```



6) Vamos criar o modelo Prato, bem como sua migration:
   a) Vamos executar o generator de model do rails novamente.
   b) `rails generate model prato` no Terminal
   c) Abra o arquivo **"db/migrate/<timestamp>_create_pratos.rb"**
   d) Adicione as linhas:

```
t.string :nome, limit: 80
```

   e) Volte para o Terminal
   f) execute rake db:migrate
   g) Olhe no console o que foi feito
   h) Olhe no banco de dados!

7) Vamos definir que o nome de um **prato** que pode ser atribuído via hash como parâmetro no construtor, edite o arquivo `app/models/prato.rb` e adicione a invocação ao `attr_accessible`:

```
attr_accessible :nome
```

8) Vamos agora fazer as validações no modelo **"prato"**. Ainda no arquivo `app/models/prato.rb` adicione o seguinte código:

```
validates_presence_of :nome, message: " - deve ser preenchido"

validates_uniqueness_of :nome, message: " - nome já cadastrado"
```

9) Vamos criar o modelo Receita, bem como sua migration:
   a) Vá ao Terminal
   b) Execute rails generate model receita
   c) Abra o arquivo **"db/migrate/<timestamp>_create_receitas.rb"**
   d) Adicione as linhas:

```
t.integer :prato_id
t.text :conteudo
```

e) Volte ao Terminal

f) Execute rake db:migrate

g) Olhe no console o que foi feito

h) Olhe no banco de dados!

10) Complete a classe `Receita` definindo que `prato_id` e `conteudo` serão passíveis de atribuição via *mass assignment*, lembra como se faz? Edite a classe `Receitas` definida em `app/models/receita.rb` e adicione a seguinte linha:

```
class Receita
  attr_accessible :prato_id, :conteudo
end
```

11) Vamos agora fazer as validações no modelo **"receita"**:

a) Abra o arquivo **"app/models/receita.rb"**

b) Adicione as seguintes linhas:

```
validates_presence_of :conteudo, message: " - deve ser preenchido"
```

7.18 Exercícios - Criando o Modelo de Qualificação

1) Vamos corrigir a pluralização da palavra 'qualificacao'

a) Abra o arquivo **"config/initializers/inflections.rb"**

b) Adicione a seguinte inflection:

```
inflect.irregular 'qualificacao', 'qualificacoes'
```

2) a) Execute o console do Rails:

```
$ rails console
```

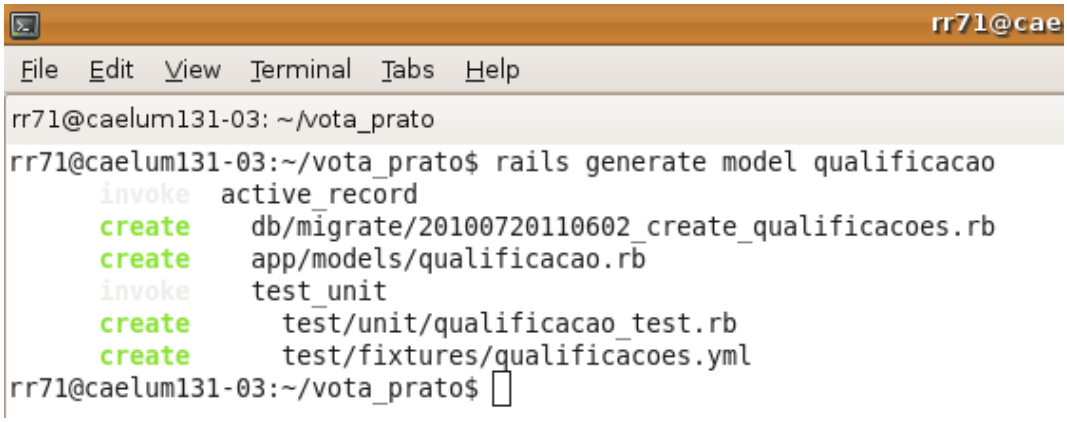
b) No Console do Rails digite:

```
"qualificacao".pluralize
```

3) Vamos continuar com a criação do nosso modelo Qualificacao e sua migration.

a) No Terminal digite

b) rails generate model qualificacao



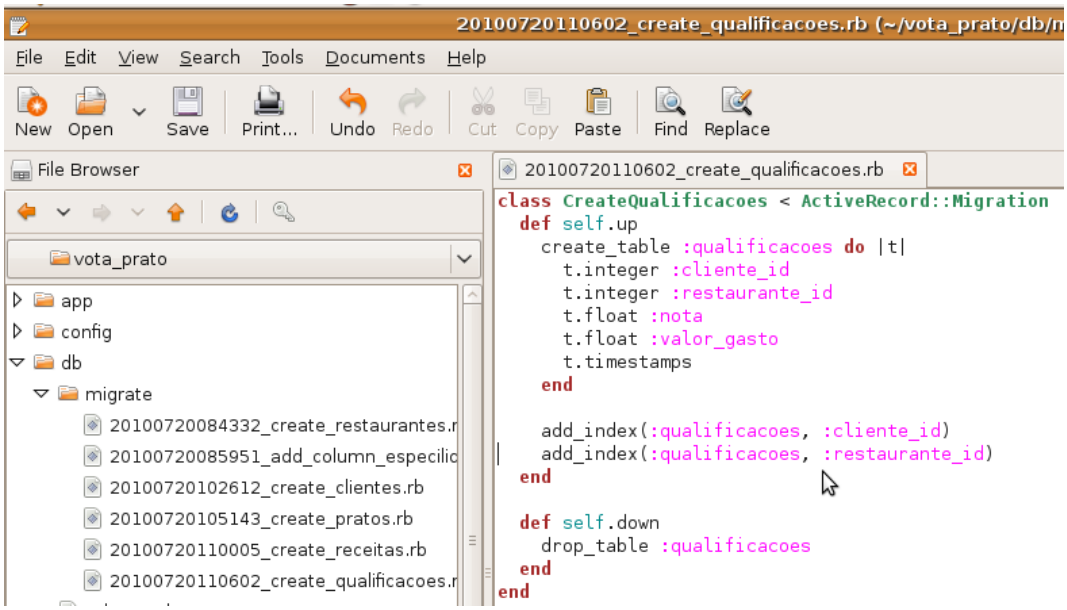
c) Abra o arquivo **"db/migrate/<timestamp>_create_qualificacoes.rb"**

d) Adicione as linhas:

```
t.integer :cliente_id
t.integer :restaurante_id
t.float :nota
t.float :valor_gasto
```

e) Adicione ainda as seguintes linhas depois de **"create_table"**: CUIDADO! Essas linhas não fazem parte do **create_table**, mas devem ficar dentro do método **change**.

```
add_index(:qualificacoes, :cliente_id)
add_index(:qualificacoes, :restaurante_id)
```

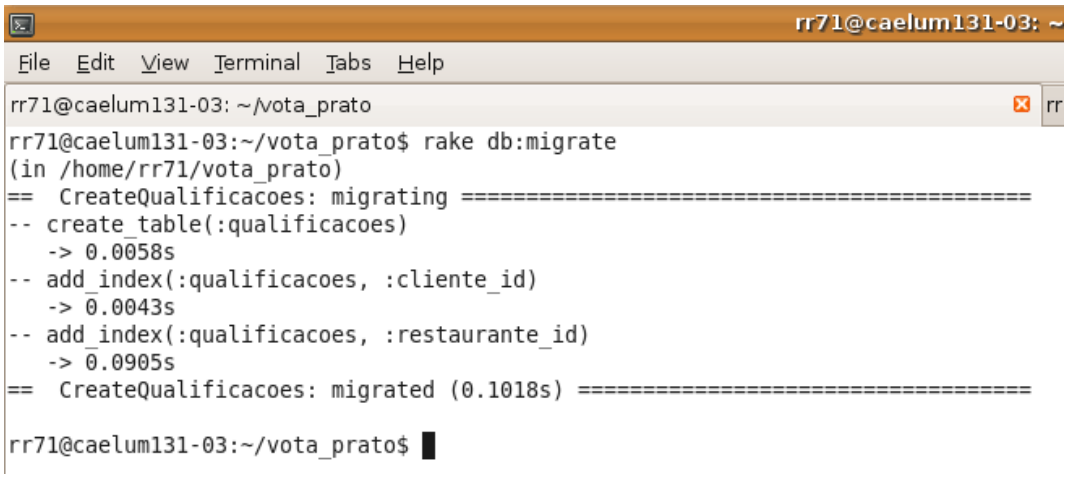


f) Volte ao Terminal

g) Execute a task para rodar as migrações pendentes:

```
$ rake db:migrate
```

h) Olhe no console o que foi feito



i) Olhe no banco de dados

```
mysql> show tables;
+-----------------------------------+
| Tables_in_vota_prato_development |
+-----------------------------------+
| clientes                          |
| pratos                            |
| qualificacoes                     |
| receitas                          |
| restaurantes                      |
| schema_migrations                 |
+-----------------------------------+
6 rows in set (0.00 sec)
```

4) Configure as propriedades de uma qualificação utilizando o `attr_accessible`:

```
attr_accessible :nota, :valor_gasto, :cliente_id, :restaurante_id
```

5) Vamos agora fazer as validações no modelo **"qualificacao"**:

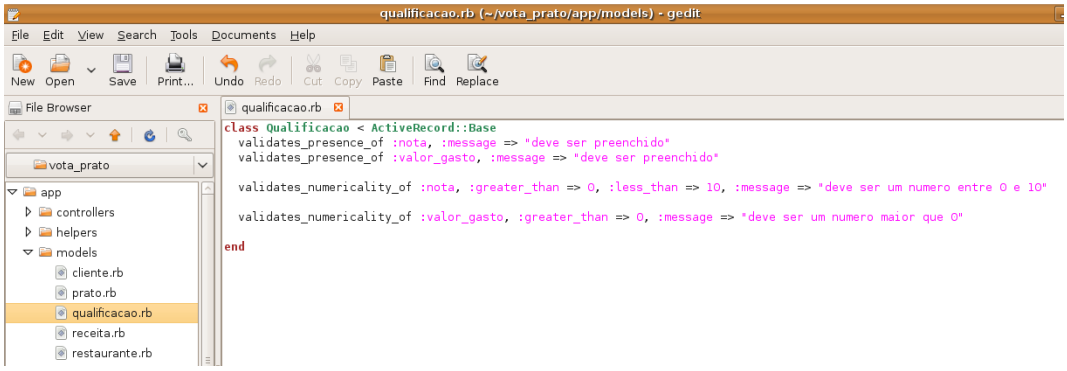
a) Abra o arquivo **"app/models/qualificacao.rb"**

b) Adicione as seguintes linhas:

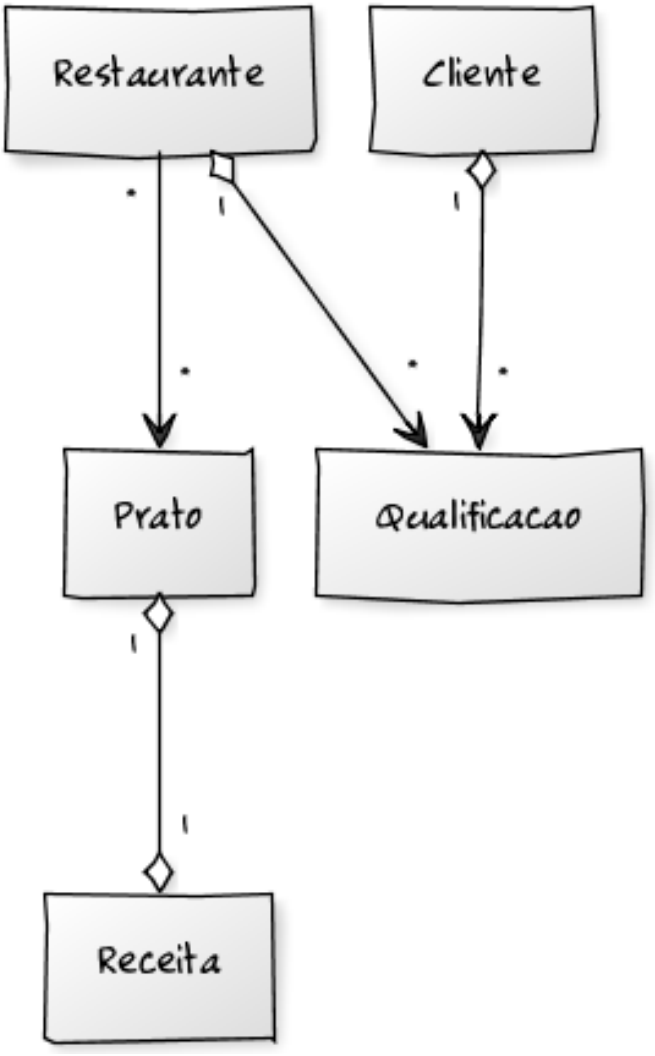
```
validates_presence_of :nota, message: " - deve ser preenchido"
validates_presence_of :valor_gasto, message: " - deve ser preenchido"

validates_numericality_of :nota, greater_than_or_equal_to: 0,
                                 less_than_or_equal_to: 10,
                                 message: " - deve ser um número entre 0 e 10"
```

```
validates_numericality_of :valor_gasto, greater_than: 0,
                                        message: " - deve ser um número maior que 0"
```



7.19 Relacionamentos



Para relacionar diversos modelos, precisamos informar ao *Rails* o tipo de relacionamento entre eles. Quando isso é feito, alguns métodos são criados para podermos manipular os elementos envolvidos nesse relacionamento. Os relacionamentos que Rails disponibiliza são os seguintes:

- belongs_to - usado quando um modelo tem como um de seus atributos o id de outro modelo (many-to-one ou one-to-one).Quando dissermos que uma qualificação **belongs_to** um restaurante, ainda ganharemos os seguintes métodos:
  - `Qualificacao.restaurante` (similar ao `Restaurante.find(restaurante_id)`)
  - `Qualificacao.restaurante=(restaurante)` (similar ao `qualificacao.restaurante_id = restaurante.id`)
  - `Qualificacao.restaurante?` (similar ao `qualificacao.restaurante == algum_restaurante`)
- has_many - associação que provê uma maneira de mapear uma relação one-to-many entre duas entidades.Quando dissermos que um restaurante **has_many** qualificações, ganharemos os seguintes métodos:
  - `Restaurante.qualificacoes` (semelhante ao `Qualificacao.find :all, conditions: ["restaurante_id = ?", id]`)
  - `Restaurante.qualificacoes<<`
  - `Restaurante.qualificacoes.delete`
  - `Restaurante.qualificacoes=`
- has_and_belongs_to_many - associação muitos-para-muitos, que é feita usando uma tabela de mapeamento.Quando dissermos que um prato **has_and_belongs_to_many** restaurantes, ganharemos os seguintes métodos:
  - `Prato.restaurantes`
  - `Prato.restaurantes<<`
  - `Prato.restaurantes.delete`
  - `Prato.restaurantes=`

  Além disso, precisaremos criar a tabela **pratos_restaurantes**, com as colunas **prato_id** e **restaurante_id**. Por convenção, o nome da tabela é a concatenação do nome das duas outras tabelas, seguindo a ordem alfabética.
- has_one - lado bidirecional de uma relação um-para-um.Quando dissermos que um prato **has_one** receita, ganharemos os seguintes métodos:
  - `Prato.receita`, (semelhante ao `Receita.find(:first, conditions: "prato_id = id")`)
  - `Prato.receita=`

7.20 Para Saber Mais: Auto-relacionamento

Os relacionamentos vistos até agora foram sempre entre dois objetos de classes diferentes. Porém existem relacionamentos entre classes do mesmo tipo, por exemplo, uma Categoria de um site pode conter outras categorias, onde cada uma contém ainda suas categorias. Esse relacionamento é chamado *auto-relacionamento*.

No ActiveRecord temos uma forma simples para fazer essa associação.

```
class Category < ActiveRecord::Base
    has_many :children, class_name: "Category",
                        foreign_key: "father_id"
    belongs_to :father, class_name: "Category"
end
```

7.21 Para Saber Mais: Cache

Todos os resultados de métodos acessados de um relacionamento são obtidos de um cache e não novamente do banco de dados. Após carregar as informações do banco, o ActiveRecord só volta se ocorrer um pedido explicito. Exemplos:

```
restaurante.qualificacoes             # busca no banco de dados
restaurante.qualificacoes.size        # usa o cache
restaurante.qualificacoes.empty?      # usa o cache
restaurante.qualificacoes(true).size  # força a busca no banco de dados
restaurante.qualificacoes             # usa o cache
```

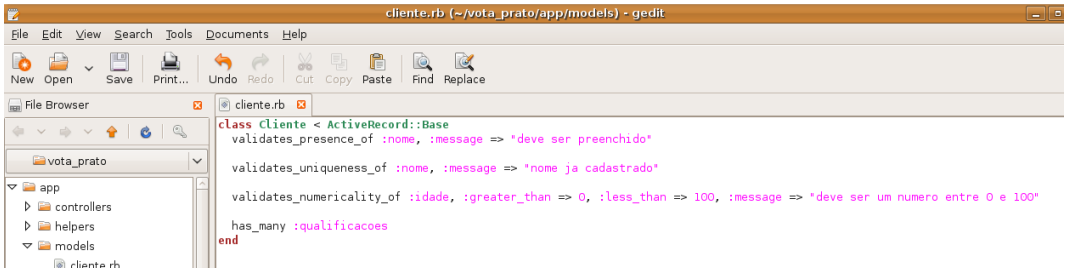
7.22 Exercícios - Relacionamentos

1) Vamos incluir os relacionamentos necessários para nossos modelos:

   a) Abra o arquivo **"app/models/cliente.rb"**

   b) Adicione a seguinte linha:

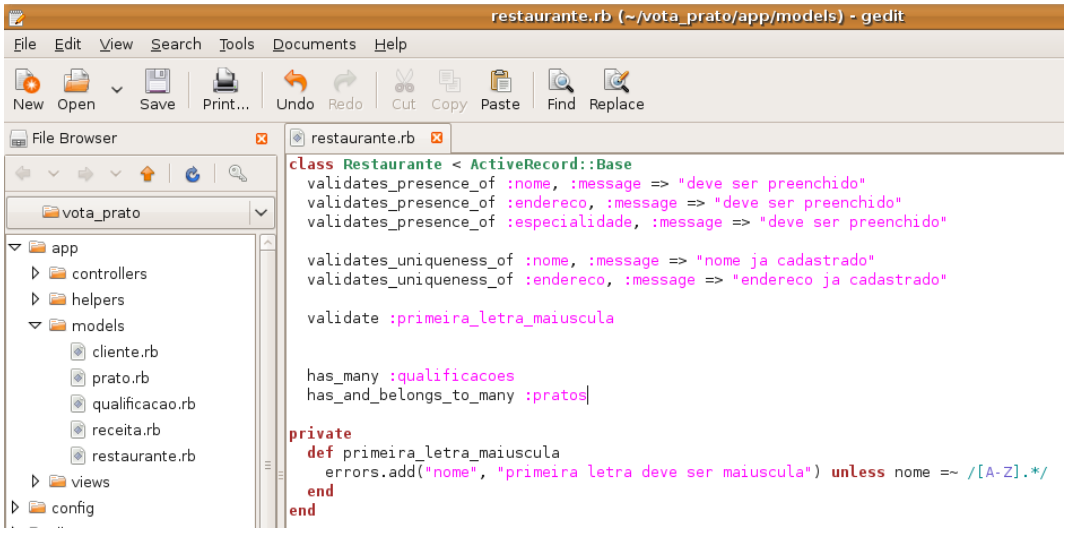
   ```
   has_many :qualificacoes
   ```



c) Abra o arquivo **"app/models/restaurante.rb"**

d) Adicione a seguinte linha:

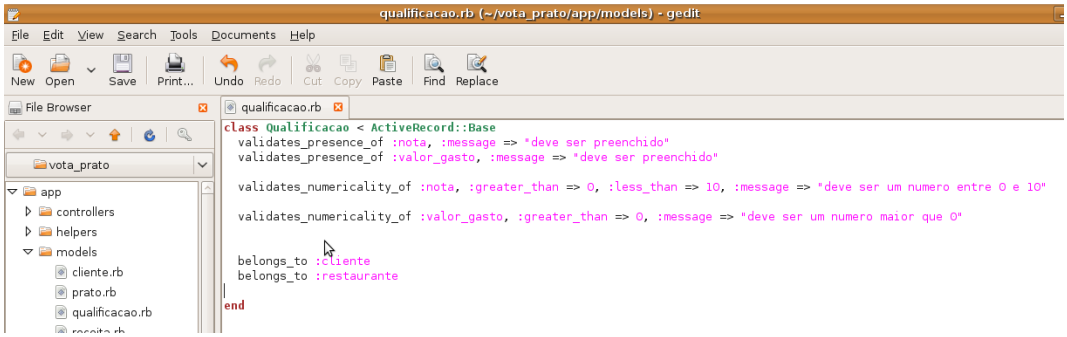
```
has_many :qualificacoes
has_and_belongs_to_many :pratos
```



e) Abra o arquivo **"app/models/qualificacao.rb"**

f) Adicione as seguintes linhas:

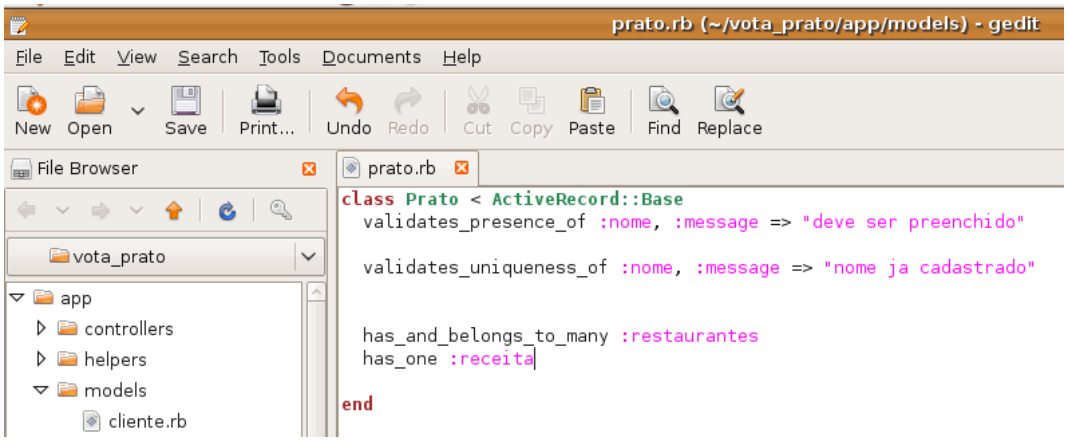
```
belongs_to :cliente
belongs_to :restaurante
```



g) Abra o arquivo **"app/models/prato.rb"**

h) Adicione as seguintes linhas:

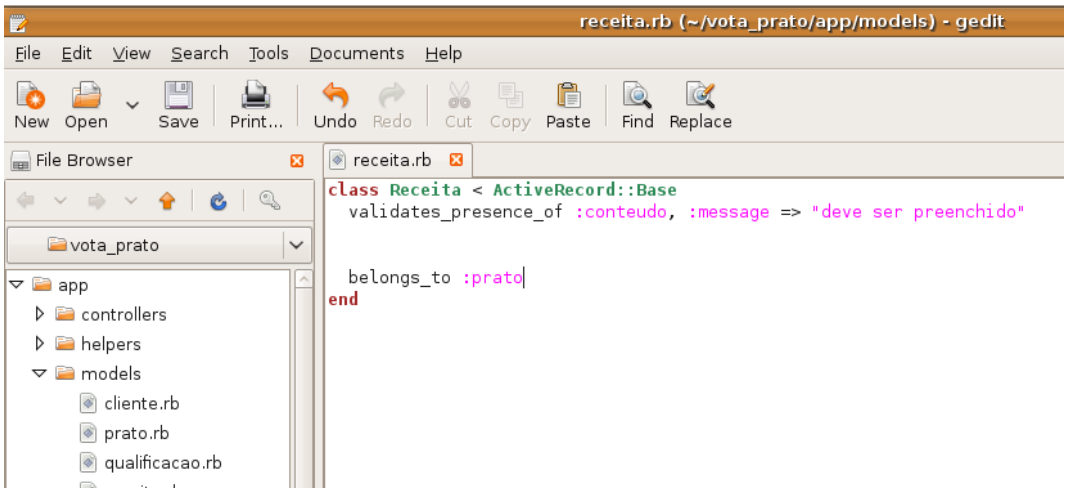
```
has_and_belongs_to_many :restaurantes
has_one :receita
```



i) Abra o arquivo **"app/models/receita"**

j) Adicione a seguinte linha:

```
belongs_to :prato
```



2) Vamos criar a tabela para nosso relacionamento **has_and_belongs_to_many**:

a) Vá para o Terminal

b) Digite: **rails generate migration createPratosRestaurantesJoinTable**

c) Abra o arquivo **"db/migrate/<timestamp>_create_pratos_restaurantes_join_table.rb"**

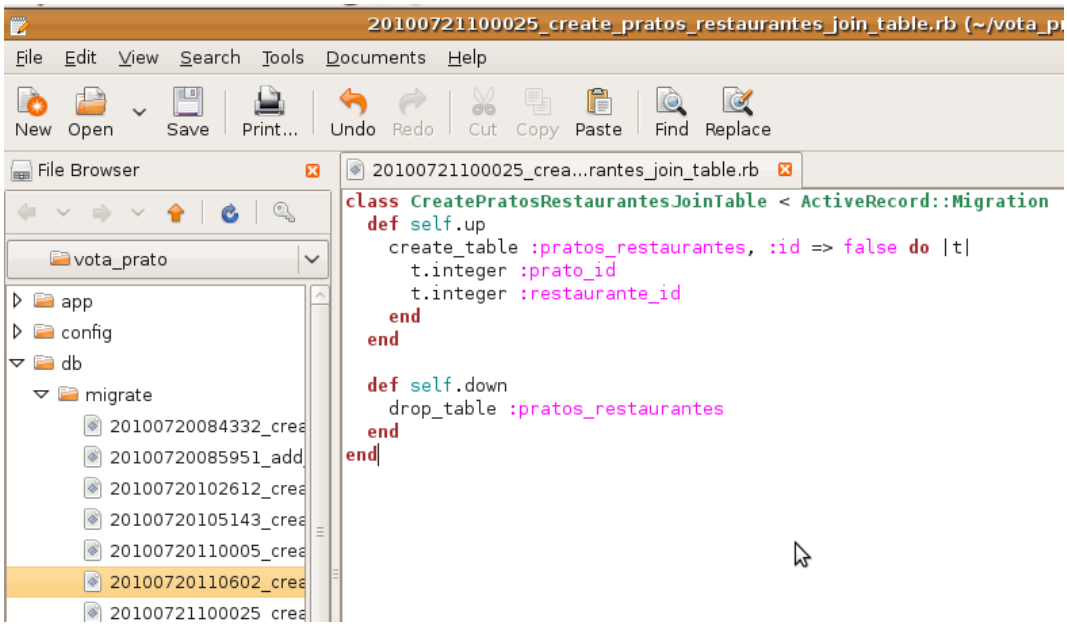
d) O rails não permite que uma tabela de ligação para um relacionamento `has_and_belongs_to_many` possua chave primária auto incremento. Por isso, vamos fazer com que o rails não gere essa chave. Basta passar o parâmetro `id: false` para o método `create_table`. Dessa forma, teremos que digitar o seguinte no método up:

```
create_table :pratos_restaurantes, id: false do |t|
  t.integer :prato_id
```

```
    t.integer :restaurante_id
  end
```

e) E o seguinte no **down**

```
drop_table :pratos_restaurantes
```



f) Volte ao Terminal e execute as migrações:

```
$ rake db:migrate
```

g) Olhe no console o que foi feito

h) Olhe no banco de dados

3) Vamos incluir as validações que garantam que os relacionamentos foram feitos corretamente:

a) Abra o arquivo **"app/models/qualificacao.rb"**

b) Adicione as seguintes linhas:

```
validates_presence_of :cliente_id, :restaurante_id
validates_associated :cliente, :restaurante
```

c) Abra o arquivo **"app/models/receita.rb"**

d) Adicione as seguintes linhas:

```
validates_presence_of :prato_id
validates_associated :prato
```

e) Abra o arquivo **"app/models/prato.rb"**

f) Adicione as seguintes linhas:

```
validate :validate_presence_of_more_than_one_restaurante

private
def validate_presence_of_more_than_one_restaurante
  errors.add("restaurantes",
        "deve haver ao menos um restaurante") if restaurantes.empty?
end
```

4) Faça alguns testes no terminal para testar essas novas validações.

7.23 Para Saber Mais - Eager Loading

Podemos cair em um problema grave caso o número de qualificações para um restaurante seja muito grande. Imagine que um determinado restaurante do nosso sistema possua com 100 qualificações. Queremos mostrar todas as qualificações desse restaurante, então fazemos um for:

```
for qualificacao in Qualificacao.all
  puts "restaurante:  " + qualificacao.restaurante.nome
  puts "cliente:      " + qualificacao.cliente.nome
  puts "qualificacao: " + qualificacao.nota
end
```

Para iterar sobre as 100 qualificações do banco de dados, seriam geradas 201 buscas! Uma busca para todas as qualificações, 100 buscas para cada restaurante mais 100 buscas para cada cliente! Podemos melhorar um pouco essa busca. Podemos pedir ao ActiveRecord que inclua o restaurante quando fizer a busca:

```
Qualificacao.find(:all, include: :restaurante)
```

Bem melhor! Agora a quantidade de buscas diminuiu para 102! Uma busca para as qualificações, outra para todos os restaurantes e mais 100 para os clientes. Podemos utilizar a mesma estratégia para otimizar a busca de clientes:

```
Qualificacao.includes(:restaurante, :cliente)
```

Com essa estratégia, teremos o número de buscas muito reduzido. A quantidade total agora será de 1 + o número de associações necessárias. Poderíamos ir mais além, e trazer uma associação de uma das associações existentes em qualificação:

```
Qualificacao.includes(:cliente, restaurante: {pratos: :receita})
```

7.24 Para Saber Mais - Named Scopes

Para consultas muito comuns, podemos usar o recurso de **Named Scopes** oferecido pelo `ActiveRecord`, que permite deixarmos alguns tipos de consultas comuns "preparadas".

Imagine que a consulta de restaurantes de especialidade "massa" seja muito comum no nosso sistema. Podemos facilitá-la criando um named scope na classe `Restaurante`:

```
class Restaurante < ActiveRecord::Base
  scope :massas, where(especialidade: 'massas')
end
```

As opções mais comuns do método find também estão disponíveis para *named scopes*, como `:conditions`, `:order`, `:select` e `:include`.

Com o *named scope* definido, a classe ActiveRecord ganha um método de mesmo nome, através do qual podemos recuperar os restaurantes de especialidade `"massas"` de forma simples:

```
Restaurante.massas
Restaurante.massas.first
Restaurante.massas.last
Restaurante.massas.where(["nome like ?", '%x%'])
```

O método associado ao *named scope* criado retorna um objeto da classe `ActiveRecord`*`NamedScope`*`Scope`, que age como um `Array`, mas aceita a chamada de alguns métodos das classes `ActiveRecord`, como o `find` para filtrar ainda mais a consulta.

Podemos ainda definir diversos *named scopes* e combiná-los de qualquer forma:

```
class Restaurante < ActiveRecord::Base
  scope :massas, where(especialidade: 'massas')
  scope :recentes, where(["created_at > ?", 3.months.ago])
  scope :pelo_nome, order('nome')
end

Restaurante.massas   # todos de especialidade = 'massas'
Restaurante.recentes # todos de created_at > 3 meses atras

# especialidade = 'massas' e created_at > 3 meses atras
Restaurante.massas.recentes
Restaurante.recentes.massas

Restaurante.massas.pelo_nome.recentes
```

7.25 Para Saber Mais - Modules

As associações procuram por relacionamentos entre classes que estejam no mesmo módulo. Caso precise de relacionamentos entre classes em módulos distintos, é necessário informar o nome completo da classe no relacionamento:

```
module Restaurante
  module RH
    class Pessoa < ActiveRecord::Base
        end
  end

  module Financeiro
    class Pagamento < ActiveRecord::Base
      belongs_to :pessoa, class_name: "Restaurante::RH::Pessoa"
    end
  end
end
```

Capítulo 8

Rotas

8.1 routes.rb

Veja abaixo como criar uma nova rota na nossa aplicação através do arquivo **config/routes.rb**:

```
VotaPrato::Application.routes.draw do
  match 'inicio', controller: 'restaurantes', action: 'index'
end
```

`match` cria uma nova rota e tem 2 formas de ser utilizado. Na nova maneira, ele recebe um único parâmetro que deve ser um hash, onde a chave do primeiro elemento é a rota que queremos mapear, e o valor para essa chave deve ser uma string com o nome do controlador e da action separado por um carácter '#', como no exemplo abaixo:

```
match 'inicio' => 'restaurantes#index'
```

Nesse exemplo o método `match` recebeu um hash onde a primeira chave é 'inicio' (rota a ser mapeada) e o valor para essa chave é 'restaurantes#index', ou seja, quando acessar `localhost:3000/inicio` o Rails vai executar a action `index` no controller `restaurantes`.

A outra abordagem é usar o método match recebendo dois parâmetros:

- url
- hash com o conjunto de parâmetros de requisição a serem preenchidos

No exemplo acima, para a url "localhost:3000/inicio" o método index do controlador de restaurantes (`RestaurantesController`) é chamado.

Qualquer parâmetro de requisição pode ser preenchido por uma rota. Tais parâmetros podem ser recuperados posteriormente através do hash de parâmetros da requisição, `params['nome']`.

Os parâmetros `:controller` e `:action` são especiais, representam o controlador e a action a serem executados.

Uma das características mais interessantes do rails é que as urls das rotas podem ser usadas para capturar alguns dos parâmetros:

```
match 'categorias/:nome', controller: 'categorias', action: 'show'
```

Neste caso, o parâmetro de requisição `params['nome']` é extraído da própria url!

> **Rotas padrão**
>
> Antigamente o Rails criava uma rota padrão. Hoje em dia ela continua no arquivo *config/routes.rb*, mas vem comentada por padrão:
>
> **match ':controller(/:action(/:id(.:format)))'**
>
> Esta rota padrão que nos permitiu usar o formato de url que temos usado até agora. O nome do controlador e a action a ser executada são retirados da própria url chamada.

Você pode definir quantas rotas forem necessárias. A ordem define a prioridade: rotas definidas no início tem mais prioridade que as do fim.

8.2 Pretty URLs

A funcionalidade de roteamento embutida no Rails é bastante poderosa, podendo até substituir mod_rewrite em muitos casos. As rotas permitem uma grande flexibilidade para criação de urls que se beneficiem de técnicas de *Search Engine Optimization* (SEO).

Um exemplo interessante seria para um sistema de blog, que permitisse a exibição de posts para determinado ano:

```
match 'blog/:ano' => 'posts#list'
```

Ou ainda para um mês específico:

```
match 'blog/:ano/:mes' => 'posts#list'
```

Os parâmetros capturados pela url podem ter ainda valores *default*:

```
match 'blog(/:ano)' => 'posts#list', :ano => 2011
```

Para o último exemplo, a url 'http://localhost:3000/blog' faria com que a action `list` do controlador `PostsController` fosse chamada, com o `params['ano']` sendo 2011.

8.3 Named Routes

Cada uma das rotas pode ter um nome único:

```
match 'blog/:ano' => 'posts#list', :as => 'posts'
```

O funcionamento é o mesmo de antes, com a diferença que usando o ':as' demos um nome à rota.

Para cada uma das *Named Routes* são criados automaticamente dois helpers, disponíveis tanto nos controladores quanto nas views:

- posts_path => '/blog/:ano'
- posts_url => 'http://localhost:3000/blog/:ano'

A convenção para o nome dos helpers é sempre `nome_da_rota_path` e `nome_da_rota_url`.

Você pode ainda ver o roteamento para cada uma das urls disponíveis em uma aplicação rails com a ajuda de uma task do rake:

```
$ rake routes
```

8.4 REST - resources

REST é um modelo arquitetural para sistemas distribuídos. A ideia básica é que existe um conjunto fixo de operações permitidas (*verbs*) e as diversas aplicações se comunicam aplicando este conjunto fixo de operações em recursos (*nouns*) existentes, podendo ainda pedir diversas representações destes recursos.

A sigla REST vem de *Representational State Transfer* e surgiu da tese de doutorado de Roy Fielding, descrevendo as ideias que levaram a criação do protocolo HTTP. A web é o maior exemplo de uso de uma arquitetura REST, onde os verbos são as operações disponíveis no protocolo (GET, POST, DELETE, PUT, HEADER, ...), os recursos são identificados pelas URLs e as representações podem ser definidas através de *Mime Types*.

Ao desenhar aplicações REST, pensamos nos recursos a serem disponibilizados pela aplicação e em seus formatos, ao invés de pensar nas operações.

Desde o Rails 1.2, o estilo de desenvolvimento REST para aplicações web é encorajado pelo framework, que possui diversas facilidades para a adoção deste estilo arquitetural.

As operações disponíveis para cada um dos recursos são:

- **GET**: retorna uma representação do recurso
- **POST**: criação de um novo recurso
- **PUT**: altera o recurso
- **DELETE**: remove o recurso

Os quatro verbos do protocolo HTTP são comumente associados às operações de CRUD em sistemas *Restful*. Há uma grande discussão dos motivos pelos quais usamos *POST* para criação (*INSERT*) e *PUT* para alteração (*UPDATE*). A razão principal é que o protocolo HTTP especifica que a operação PUT deve ser *idempotente*, já POST não.

> **Idempotência**
>
> Operações idempotentes são operações que podem ser chamadas uma ou mais vezes, sem diferenças no resultado final. Idempotência é uma propriedade das operações.

A principal forma de suporte no Rails a estes padrões é através de rotas que seguem as convenções da arquitetura REST. Ao mapear um recurso no routes.rb, o Rails cria automaticamente as rotas adequadas no controlador para tratar as operações disponíveis no recurso (GET, POST, PUT e DELETE).

```
# routes.rb
resources :restaurantes
```

Ao mapear o recurso `:restaurantes`, o rails automaticamente cria as seguintes rotas:

- GET /restaurantes: `controller => 'restaurantes', :action => 'index'`
- POST /restaurantes: `controller => 'restaurantes', :action => 'create'`
- GET /restaurantes/new: `controller => 'restaurantes', :action => 'new'`
- GET /restaurantes/:id: `controller => 'restaurantes', :action => 'show'`
- PUT /restaurantes/:id: `controller => 'restaurantes', :action => 'update'`
- DELETE /restaurantes/:id: `controller => 'restaurantes', :action => 'destroy'`
- GET /restaurantes/:id/edit: `controller => 'restaurantes', :action => 'edit'`

Como é possível perceber através das rotas, todo recurso mapeado implica em sete métodos no controlador associado. São as famosas sete actions REST dos controladores rails.

Além disso, para cada rota criada, são criados os helpers associados, já que as rotas são na verdade *Named Routes*.

```
restaurantes_path          # => "/restaurantes"
new_restaurante_path       # => "/restaurantes/new"
edit_restaurante_path(3)   # => "/restaurantes/3/edit"
```

Rails vem com um `generator` pronto para a criação de novos recursos. O controlador (com as sete actions), o modelo, a migration, os esqueleto dos testes (unitário, funcional e fixtures) e a rota podem ser automaticamente criados.

```
$ rails generate resource comentario
```

O gerador de scaffolds do Rails 2.0 em diante, também usa o modelo REST:

```
$ rails generate scaffold comentario conteudo:text author:string
```

Na geração do scaffold são produzidos os mesmos artefatos de antes, com a adição das views e de um layout padrão.

Não deixe de verificar as rotas criadas e seus nomes (*Named Routes*):

```
$ rake routes
```

8.5 Actions extras em Resources

As sete actions disponíveis para cada resource costumam ser suficientes na maioria dos casos. Antes de colocar alguma action extra nos seus resources, exercite a possibilidade de criar um novo resource para tal.

Quando necessário, você pode incluir algumas actions extras para os resources:

```
resources :comentarios do
  member do
    post :desabilita
  end
# url: /comentarios/:id/desabilita
# named_route: desabilita_comentario_path
end
```

`:member` define actions que atuam sobre um recurso específico: */comentarios/1/desabilita*. Dentro do bloco member usar dessa forma `method :action`, onde `method` pode ser `get`, `post`, `put`, `delete` ou `any`.

Outro bloco que pode ser usado dentro de um resource é o `collection` que serve para definir actions extras que atuem sobre o conjunto inteiro de resources. Definirá rotas do tipo */comentarios/action*.

```
resources :comentarios do
  collection do
    get :feed
  end
# url: /comentarios/feed
# named_route: feed_comentarios_path
end
```

Para todas as actions extras, são criadas *Named Routes* adequadas. Use o `rake routes` como referência para conferir os nomes dados às rotas criadas.

8.6 Para Saber Mais - Nested Resources

Quando há relacionamentos entre resources, podemos aninhar a definição das rotas, que o rails cria automaticamente as urls adequadas.

No nosso exemplo, :restaurante has_many :qualificacoes, portanto:

```
# routes.rb
resources :restaurantes do
  resources :qualificacoes
end
```

A rota acima automaticamente cria as rotas para qualificações específicas de um restaurante:

- GET /restaurante/:restaurante_id/qualificacoes:controller => 'qualificacoes', :action => 'index'
- GET /restaurante/:restaurante_id/qualificacoes/:id:controller => 'qualificacoes', :action => 'show'
- GET /restaurante/:restaurante_id/qualificacoes/new:controller => 'qualificacoes', :action => 'new'
- ...

As sete rotas comuns são criadas para o recurso `:qualificacao`, mas agora as rotas de `:qualificacoes` são sempre específicas a um `:restaurante` (todos os métodos recebem o `params['restaurante_id']`).

Capítulo 9

Controllers e Views

"A Inteligência é quase inútil para quem não tem mais nada"
– Carrel, Alexis

Nesse capítulo, você vai aprender o que são controllers e como utilizá-los para o benefício do seu projeto, além de aprender a trabalhar com a camada visual de sua aplicação.

9.1 O "V" e o "C" do MVC

O "V" de MVC representa a parte de **view** (visualização) da nossa aplicação, sendo ela quem tem contato com o usuário, recebe as entradas e mostra qualquer tipo de saída.

Há diversas maneiras de controlar as views, sendo a mais comum delas feita através dos arquivos HTML.ERB, ou eRuby (Embedded Ruby), páginas HTML que podem receber trechos de código em Ruby.

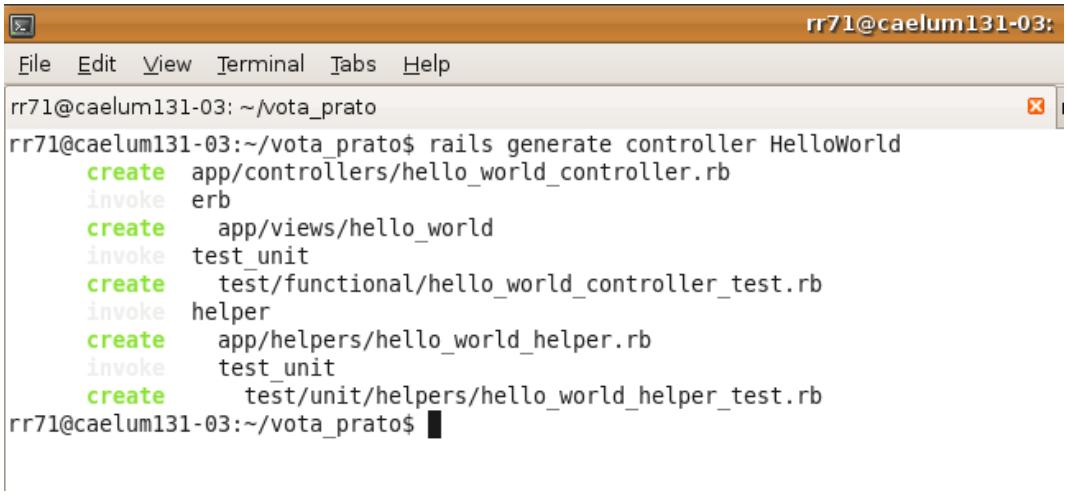
Controllers são classes que recebem uma ação de uma View e executam algum tipo de lógica ligada a um ou mais modelos. Em Rails esses controllers estendem a classe `ApplicationController`.

As urls do servidor são mapeadas da seguinte maneira: **/controller/action/id**. Onde "controller" representa uma classe controladora e "action" representa um método do mesmo. "id" é um parâmetro qualquer (opcional).

9.2 Hello World

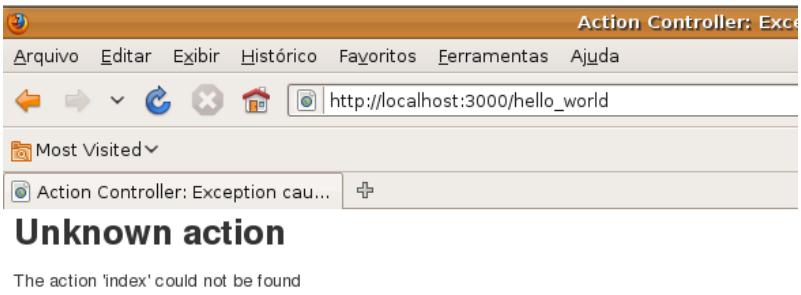
Antes de tudo criaremos um controller que mostrará um *"Hello World"* para entender melhor como funciona essa ideia do mapeamento de urls.

Vamos usar o generator do rails para criar um novo controller que se chamará *"HelloWorld"*. Veja que o Rails não gera apenas o Controller, mas também outros arquivos.



Apos habilitar o rota padrão do rails tente acessar a página http://localhost:3000/hello_world

Na URL acima passamos apenas o controller sem nenhuma action, por isso recebemos uma mensagem de erro.



Além de não dizer qual a action na URL, não escrevemos nenhuma action no controller.

Criaremos um método chamado `hello` no qual escreveremos na saída do cliente a frase "Hello World!". **Cada método criado no controller é uma action**, que pode ser acessada através de um browser.

Para escrever na saída, o Rails oferece o comando `render`, que recebe uma opção chamada "text" (String). Tudo aquilo que for passado por esta chave será recebido no browser do cliente.

9.3 Exercícios: Criando o controlador

1) Vamos criar um controller para nossos restaurantes:

a) Abra o Terminal.

b) Execute **rails generate controller restaurantes**

2) Mais tarde esse controller irá mediar todas as interações relacionadas à restaurantes. Agora ele irá simplesmente renderizar um HTML com o texto "Estou no controller de restaurantes!" envolvido por um parágrafo.

a) Crie um arquivo HTML para o index: **app/views/restaurantes/index.html.erb**

b) Como conteúdo do arquivo digite:

```
<p>
  Estou no controller de Restaurantes!
</p>
```

3) a) Abra seu novo controller (**app/controllers/restaurantes_controller.rb**)

b) Inclua a action **index**:

```
def index
  render "index"
end
```

4) Vamos habilitar as rotas padrões para restaurantes:

a) Abra o arquivo **config/routes.rb**.

b) Dentro do bloco de código adicione a linha:

```
resources :restaurantes
```

5) a) O último passo antes de testar no browser é iniciar o server.

b) Execute no Terminal: **rails server**

c) Confira o link http://localhost:3000/restaurantes.

9.4 Trabalhando com a View: O ERB

ERB

ERB é uma implementação de eRuby que já acompanha a linguagem Ruby. Seu funcionamento é similar ao dos arquivos JSP/ASP: arquivos html com injeções de código. A ideia é que o HTML serve como um template, e outros elementos são dinamicamente inseridos em tempo de renderização.

SINTAXE BÁSICA

Para uma página aceitar código Ruby, ela deve estar entre "<%" e "%>". Há uma variação deste operador, o "<%=", que não só executa códigos Ruby, mas também imprime o resultado na página HTML.

Logo, o seguinte código ERB:

```
<html>
  <body>
    <p>Meu nome é: <%= "João" %></p>
    <p>Não vai imprimir nada ao lado: <% "Não será impresso" %></p>
  </body>
</html>
```

Irá resultar em:

```
<html>
  <body>
    <p>Meu nome é: João</p>
    <p>Não vai imprimir nada ao lado: </p>
  </body>
</html>
```

IF, ELSE E BLOCOS

O operador "<%" é muito útil quando precisamos que um pedaço de HTML seja adicionado com uma condição. Por exemplo:

```
<body>
  <% if nomes.empty? %>
    <div class="popup">
      Nenhum nome
    </div>
  <% else %>
    <div class="listagem">
      <%= nomes %>
    </div>
  <% end %>
</body>
```

Caso a variável nomes seja igual à [], o resultado será:

```
<body>
    <div class="popup">
```

```
      Nenhum nome
    </div>
</body>
```

Por outro lado, se `nomes` for igual à `["João", "Maria"]` o resultado será:

```
<body>
    <div class="listagem">
      ["João", "Maria"]
    </div>
</body>
```

Podemos ainda iterar pelos nomes imprimindo-os em uma lista, basta mudar o ERB para:

```
<body>
  <% if nomes.empty? %>
    <div class="popup">
      Nenhum nome
    </div>
  <% else %>
    <ul class="listagem">
      <% nomes.each do |nome| %>
        <li><%= nome %></li>
      <% end %>
    </ul>
  <% end %>
</body>
```

Para `nomes` igual à `["João", "Maria"]` o resultado seria:

```
<body>
    <ul class="listagem">
        <li><%= João %></li>
        <li><%= Maria %></li>
    </ul>
</body>
```

Do controller para view

É importante notar que todos os atributos de instância (`@variavel`) de um controlador estão disponíveis em sua view. Além disso, ela deve ter o mesmo nome da action, e estar na pasta com o nome do controlador o que significa que a view da nossa action `index` do controlador `restaurantes_controller.rb` deve estar em: **app/views/restaurantes/index.html.erb**.

9.5 Entendendo melhor o CRUD

Agora, queremos ser capazes de criar, exibir, editar e remover restaurantes. Como fazer?

Primeiro, temos de criar um controller para nosso restaurante. Pela view Generators, vamos criar um controller para restaurante. Rails, por padrão, utiliza-se de sete actions "CRUD". São eles:

- index: exibe todos os items
- show: exibe um item específico
- new: formulário para a criação de um novo item
- create: cria um novo item
- edit: formulário para edição de um item
- update: atualiza um item existente
- destroy: remove um item existente

9.6 A action index

Desejamos listar todos os restaurantes do nosso Banco de Dados, e portanto criaremos a action `index`.

Assim como no console buscamos todos os restaurantes do banco de dados com o comando `find`, também podemos fazê-lo em controllers (que poderão ser acessados pelas nossas views, como veremos mais adiante).

Basta agora passar o resultado da busca para uma variável:

```
def index
  @restaurantes = Restaurante.order :nome
end
```

Com a action pronta, precisamos criar a view que irá utilizar a variável `@restaurantes` para gerar o HTML com a listagem de restaurantes:

```
<table>
  <tr>
    <th>Nome</th>
    <th>Endereço</th>
    <th>Especialidade</th>
  </tr>
<% @restaurantes.each do |restaurante| %>
  <tr>
    <td><%= restaurante.nome          %></td>
```

```
    <td><%= restaurante.endereco      %></td>
    <td><%= restaurante.especialidade %></td>
  </tr>
<% end %>
</table>
```

De acordo com a convenção, esta view deve estar no arquivo: **app/views/restaurantes/index.html.erb**.

Agora, basta acessar nossa nova página através da URL: **http://localhost:3000/restaurantes/**.

9.7 Exercícios: Listagem de restaurantes

Vamos agora criar uma forma do usuário visualizar uma listagem com todos os restaurantes:

1) Gere um controller para o modelo restaurante:
   a) Vá ao Terminal;
   b) Execute **rails generate controller restaurantes**
2) a) Abra o seu controller de Restaurantes (**app/controllers/restaurantes_controllers.rb**)
   b) Adicione a action index:

```
def index
  @restaurantes = Restaurante.order :nome
end
```

3) a) Vamos criar também o arquivo ERB (**app/views/restaurantes/index.html.erb**) que servirá de base para a página HTML.

```
<h1>Listagem de Restaurantes</h1>

<table>
  <tr>
    <th>ID</th>
    <th>Nome</th>
    <th>Endereço</th>
    <th>Especialidade</th>
  </tr>

<% @restaurantes.each do |restaurante| %>
  <tr>
    <td><%= restaurante.id %></td>
    <td><%= restaurante.nome %></td>
    <td><%= restaurante.endereco %></td>
    <td><%= restaurante.especialidade %></td>
```

```
    </tr>
  <% end %>
  </table>
```

4) a) Teste agora entrando em: **http://localhost:3000/restaurantes** (Não esqueça de iniciar o servidor)

   b) É possível que não tenhamos nenhum restaurante em nossa base de dados, nesse caso, vamos usar um pouco do poder de ruby para rapidamente criar alguns restaurantes via console (`rails console`):

```
especialidades = %w{massas japonês vegetariano}
50.times do |i|
  Restaurante.create!(
    nome: "Restaurante #{i}",
    endereco: "Rua #{i} de setembro",
    especialidade: especialidades.sample
  )
end
```

9.8 Helper

Helpers são módulos que disponibilizam métodos para serem usados em nossas views. Eles provêm atalhos para os códigos mais usados e nos poupam de escrever muito código. O propósito de um helper é simplificar suas views.

Helpers padrões

O Rails já nos disponibiliza uma série de helpers padrões, por exemplo, se quisermos criar um link, podemos usar o helper `link_to`:

```
<%= link_to "Restaurantes", controller: "restaurantes", action: "index" %>
```

Abaixo, uma lista com alguns helpers padrões:

- link_to (âncora)
- image_tag (img)
- favicon_link_tag (link para favicon)
- stylesheet_link_tag (link para CSS)
- javascript_include_tag (script)

Helper Method

Existe também o chamado `helper_method`, que permite que um método de seu controlador vire um Helper e esteja disponível na view para ser chamado. Exemplo:

```
class TesteController < ApplicationController
  helper_method :teste

  def teste
    "algum conteudo dinâmico"
  end
end
```

E em alguma das views deste controlador:

```
<%= teste %>
```

9.9 A action show

Para exibir um restaurante específico, precisamos saber qual restaurante buscar. Pela convenção, poderíamos visualizar o restaurante de id 1 acessando a URL: **http://localhost:3000/restaurantes/1**. Logo, o id do restaurante que buscamos deve ser passado na URL.

Ou seja, o caminho para a action show é: `/restaurantes/:id`. A action show receberá um parâmetro `:id` ao ser evocada. E esse id será utilizado na busca pelo restaurante:

```
def show
  @restaurante = Restaurante.find(params[:id])
end
```

Após criar a action, devemos criar a view que irá descrever um restaurante:

```
<h1>Exibindo Restaurante</h1>

Nome: <%= @restaurante.nome %>
Endereço: <%= @restaurante.endereco %>
Especialidade: <%= @restaurante.especialidade %>
```

Agora, basta acessar nossa nova página através da URL: **http://localhost:3000/restaurantes/id-do-restaurante**.

9.10 Exercícios: Visualizando um restaurante

1) a) Abra o seu controller de Restaurantes (**app/controllers/restaurantes_controllers.rb**)

   b) Adicione a action show:

```
def show
  @restaurante = Restaurante.find(params[:id])
end
```

2) Vamos criar também o arquivo ERB (**app/views/restaurantes/show.html.erb**).

```
<h1>Exibindo Restaurante</h1>

<p>
  <b>Nome: </b>
  <%= @restaurante.nome %>
</p>

<p>
  <b>Endereço: </b>
  <%= @restaurante.endereco %>
</p>

<p>
  <b>Especialidade: </b>
  <%= @restaurante.especialidade %>
</p>
```

3) Vamos agora usar o helper de âncora (**link_to**) para criar o link que tornará possível visualizar um restaurante apartir da listagem de restaurantes.

   a) Abra a view de index (**app/views/restaurantes/index.html.erb**).

   b) Vamos adicionar um link **Mostrar** ao lado de cada restaurante, que irá direcionar o usuário para o show do restaurante. Para isso, vamos substituir esse código:

```
<% @restaurantes.each do |restaurante| %>
  <tr>
    <td><%= restaurante.nome %></td>
    <td><%= restaurante.endereco %></td>
    <td><%= restaurante.especialidade %></td>
  </tr>
<% end %>
```

   Por esse:

```
<% @restaurantes.each do |restaurante| %>
  <tr>
```

```
        <td><%= restaurante.nome %></td>
        <td><%= restaurante.endereco %></td>
        <td><%= restaurante.especialidade %></td>
        <td>
          <%= link_to 'Mostrar', action: 'show', id: restaurante %>
        </td>
      </tr>
    <% end %>
```

4) Teste clicando no link **Mostrar** de algum restaurante na listagem de restaurantes (**http://localhost:3000/restaurantes**).

9.11 A action destroy

Para remover um restaurante, o usuário enviará uma requisição à nossa action `destroy` passando no id da url o id do restaurante a ser excluído:

```
def destroy
  @restaurante = Restaurante.find(params[:id])
  @restaurante.destroy
end
```

Observe que a action destroy não tem intenção de mostrar nenhuma informação para o usuário. Por isso, após finalizar seu trabalho, ela irá chamar a index. Abaixo iremos entender melhor o mecanismo que iremos utilizar para implementar isso.

9.12 Redirecionamento de Action

Dentro de uma action, podemos redirecionar a saída para uma outra action.

Por exemplo, se tivermos um controller qualquer com duas actions podemos utilizar o método `redirect_to` para que o usuário seja redirecionado para a **action1** no momento que ele acessar a **action2**:

```
class ExemploController < ApplicationController
  def action1
    render text: "ACTION 1!!!"
  end

  def action2
    redirect_to action: 'action1'
  end
end
```

Dessa forma, é possível que uma action exerça sua responsabilidade e depois delegue o final do processo para alguma outra action.

Imagine no nosso controller de restaurantes, por exemplo. Criamos uma action para apagar um restaurante, porém o que acontecerá após o restaurante ser deletado?

Faz sentido que o usuário volte para a página de index, para obtermos esse comportamento utilizaremos do `redirect_to`:

```
def index
  @restaurantes = Restaurante.all
end

def destroy
  # código que deleta o restaurante

  redirect_to action: 'index'
end
```

> **Redirecionamento no servidor e no cliente**
>
> O redirecionamento no servidor é conhecido como *forward* e a requisição é apenas repassada a um outro recurso (página, controlador) que fica responsável em tratar a requisição.
>
> Há uma outra forma que é o **redirecionamento no cliente** (*redirect*). Nesta modalidade, o servidor responde a requisição original com um **pedido de redirecionamento**, fazendo com que o navegador dispare uma nova requisição para o novo endereço. Neste caso, a barra de endereços do navegador muda.

9.13 Exercícios: Deletando um restaurante

1) a) Abra o seu controller de Restaurantes (**app/controllers/restaurantes_controllers.rb**)

   b) Adicione a action destroy:

```
def destroy
  @restaurante = Restaurante.find(params[:id])
  @restaurante.destroy

  redirect_to(action: "index")
end
```

2) a) Abra a view de index (**app/views/restaurantes/index.html.erb**).

   b) Vamos adicionar um link **Deletar** ao lado de cada restaurante, que irá invocar a action destroy:

```
<!-- Deve ficar abaixo do td que criamos para o link "Mostrar" -->
<td>
  <%= link_to 'Deletar', {action: 'destroy', id: restaurante},
                         {method: "delete"}  %>
</td>
```

3) Teste clicando no link **Deletar** de algum restaurante na listagem de restaurantes (**http://localhost:3000/restaurantes**).

9.14 Helpers para formulários

Quando trabalhamos com formulários, usamos os chamados **FormHelpers**, que são módulos especialmente projetados para nos ajudar nessa tarefa. Todo **FormHelper** está associado a um ActiveRecord. Existem também os **FormTagHelpers**, que contém um *_tag* em seu nome. **FormTagHelpers**, não estão necessariamente associados a ActiveRecord algum.

Abaixo, uma lista dos `FormHelpers` disponíveis:

- check_box
- fields_for
- file_field
- form_for
- hidden_field
- label
- password_field
- radio_button
- text_area
- text_field

E uma lista dos `FormTagHelpers`:

- check_box_tag
- field_set_tag
- file_field_tag
- form_tag
- hidden_field_tag

- image_submit_tag
- password_field_tag
- radio_button_tag
- select_tag
- submit_tag
- text_area_tag
- text_field_tag

Agora, podemos escrever nossa view:

```
<%= form_tag :action => 'create' do %>
  Nome: <%= text_field :restaurante, :nome %>
  Endereço: <%= text_field :restaurante, :endereco %>
  Especialidade: <%= text_field :restaurante, :especialidade %>
  <%= submit_tag 'Criar' %>
<% end %>
```

Este ERB irá renderizar um HTML parecido com:

```
<form action="/restaurantes" method="POST">
  Nome: <input type="text" name="restaurante[nome]">
  Endereço: <input type="text" name="restaurante[endereco]">
  Especialidade: <input type="text" name="restaurante[especialidade]">
  <input type="submit">
<% end %>
```

Repare que como utilizamos o `form_tag`, que não está associado a nenhum ActiveRecord, nosso outro Helper **text_field** não sabe qual o ActiveRecord que estamos trabalhando, sendo necessário passar para cada um deles o parâmetro `:restaurante`, informando-o.

Podemos reescrever mais uma vez utilizando o FormHelper `form_for`, que está associado a um `ActiveRecord`:

```
<%= form_for @restaurante do |f| %>
  Nome: <%= f.text_field :nome %>
  Endereço: <%= f.text_field :endereco %>
  Especialidade: <%= f.text_field :especialidade %>
  <%= f.submit %>
<% end %>
```

Repare agora que não foi preciso declarar o nome do nosso modelo para cada `text_field`, uma vez que nosso Helper `form_for` já está associado a ele.

9.15 A action new

Para incluir um novo restaurante, precisamos primeiro retornar ao browser um restaurante novo, sem informação alguma. Vamos criar nossa action `new`

```
def new
 @restaurante = Restaurante.new
end
```

9.16 Exercícios: Página para criação de um novo restaurante

1) a) Abra o seu controller de Restaurantes (**app/controllers/restaurantes_controllers.rb**)

   b) Adicione a action new:

```
def new
  @restaurante = Restaurante.new
end
```

2) a) Vamos criar também o arquivo ERB (**app/views/restaurantes/new.html.erb**).

```
<h1>Adicionando Restaurante</h1>

<%= form_for @restaurante do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= f.submit %>
<% end %>
```

3) a) Abra a view de index (**app/views/restaurantes/index.html.erb**).

   b) Vamos adicionar um link **Novo** para direcionar o usuário para a action new. Diferente dos outros links que criamos, esse deverá ser inserido abaixo da lista de restaurantes:

```
<!-- lista de restaurantes -->

<br/>
<%= link_to 'Novo', action: 'new' %>
```

4) Teste clicando no link **Novo** na listagem de restaurantes (**http://localhost:3000/restaurantes**).

9.17 Recebendo um parâmetro por um formulário

Não são raros os momentos onde precisamos que o usuário digite informações que utilizaremos no lado do servidor. Para isso utilizamos formulários como:

```
<form action='/restaurantes'>
  Nome: <input type='text' name='nome'/>
  <input type='submit' value='Create'/>
</form>
```

Porém, como teremos acesso ao texto digitado pelo usuário? Toda informação introduzida em um formulário, é passada para o controller como um parâmetro. Logo, para receber este valor `nome` no controlador, basta usar o hash `params`. (Repare que agora usamos outra action, `create`, para buscar os dados do formulário apresentado anteriormente):

```
class RestaurantesController < ApplicationController
  def create
    nome = params[:nome]
  end
end
```

Porém essa não é a action create dos restaurantes, precisamos de algo mais completo.

Recebendo parâmetros ao utilizar o helper form_for

Diferente do exemplo anterior, nossa view `new.html.erb` utiliza o helper `form_for` para representar o formulário de um restaurante, da seguinte maneira:

```
<%= form_for @restaurante do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= f.submit %>
<% end %>
```

Ao utilizar o helper `form_for` o Rails irá agrupar os parâmetros desse formulário na chave `:restaurante`. Logo, é possível imprimir cada um dos parâmetros da seguinte maneira:

```
def create
  puts params[:restaurante][:nome]
  puts params[:restaurante][:endereco]
  puts params[:restaurante][:especialidade]
end
```

Porém, imprimir os parâmetros não é suficiente para persistir um restaurante.

9.18 A action create

O agrupamento de parâmetros mostrado anteriormente combina perfeitamente com a hash aceita pelas classes do ActiveRecord, por isso podemos criar um novo restaurante da seguinte maneira:

```
Restaurante.new params[:restaurante]
```

Não faz sentido que criemos um novo restaurante sem persisti-lo. Vamos criar uma variável e utilizá-la em seguida para salvá-lo:

```
def create
  @restaurante = Restaurante.new params[:restaurante]
  @restaurante.save
end
```

9.19 Exercícios: Persistindo um restaurante

1) a) Abra o seu controller de Restaurantes (**app/controllers/restaurantes_controllers.rb**)

   b) Adicione a action create:

```
def create
  @restaurante = Restaurante.new(params[:restaurante])
  @restaurante.save
  redirect_to(action: "show", id: @restaurante)
end
```

9.20 A action edit

Para editar um restaurante, devemos retornar ao browser o restaurante que se quer editar, para só depois salvar as alterações feitas:

```
def edit
  @restaurante = Restaurante.find(params[:id])
end
```

A view de **edit.html.erb** terá o mesmo formulário utilizado na view **new.html.erb**:

```
<%= form_for @restaurante do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
```

```
  <%= f.submit %>
<% end %>
```

Perceba que ao editar, é usual que o usuário possa ver as informações atuais do restaurante. Para isso, não precisaremos fazer nada pois o helper `form_for` irá se encarregar de restaurar os dados do restaurante no formulário.

Após implementar a action e a view, podemos acessar o formulário de edição acessando a URL: **http://localhost:3000/restaurantes/id-do-restaurante/edit**.

9.21 A action update

Uma vez que o usuário tenha atualizado as informações do restaurante e deseje salvá-las, enviará uma requisição à nossa action `update` passando o id do restaurante a ser editado na url, bem como os novos dados do restaurante.

A hash **params** terá uma estrutura parecida com:

```
{
  id: 1,
  restaurante: {
    nome: "Comidubom",
    endereco: "Rua da boa",
    especialidade: "massas"
  }
}
```

Logo, podemos implementar nosso **update** da seguinte maneira:

```
def update
  @restaurante = Restaurante.find params[:id]
  @restaurante.update_attributes params[:restaurante]
end
```

9.22 Atualizando um restaurante

Vamos implementar as actions e views envolvidas na atualização de um restaurante. Ao final, note como o processo de atualização tem várias semelhanças com o de criação.

1) a) Abra o seu controller de Restaurantes (**app/controllers/restaurantes_controllers.rb**)

   b) Adicione a action edit:

```
def edit
  @restaurante = Restaurante.find params[:id]
end
```

2) a) Vamos criar também o arquivo ERB (**app/views/restaurantes/edit.html.erb**).

```
<h1>Editando Restaurante</h1>

<%= form_for @restaurante do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= f.submit %>
<% end %>
```

3) a) Abra o seu controller de Restaurantes (**app/controllers/restaurantes_controllers.rb**)

b) Adicione a action update:

```
def update
  @restaurante = Restaurante.find(params[:id])
  @restaurante.update_attributes(params[:restaurante])

  redirect_to action: "show", id: @restaurante
end
```

4) a) Abra a view de index (**app/views/restaurantes/index.html.erb**).

b) Vamos adicionar um link **Editar** ao lado de cada restaurante, que irá direcionar o usuário para a action edit:

```
<!-- Deve ficar abaixo do td que criamos para o link "Deletar" -->

<td>
  <%= link_to 'Editar', action: 'edit', id: restaurante %>
</td>
```

5) Teste clicando no link **Editar** na listagem de restaurantes (**http://localhost:3000/restaurantes**).

9.23 Exercícios Opcionais: linkando melhor as views

Até agora só criamos links na página que lista os restaurantes, como se fosse uma espécie de menu principal. Porém, na grande maioria dos casos, criamos links nas outras páginas também. Por exemplo, se o usuário estiver visualizando um restaurante, seria conveniente ele ter um link para poder voltar à listagem. Vamos criar esses links agora:

1) a) Abra o arquivo ERB da action new (**app/views/restaurantes/new.html.erb**).

b) Adicione no final do ERB o seguinte link:

```
<%= link_to 'Voltar', {action: 'index'} %>
```

2) a) Abra o arquivo ERB da action edit (**app/views/restaurantes/edit.html.erb**).

b) Adicione no final do ERB o seguinte link:

```
<%= link_to 'Voltar', action: 'index' %>
```

3) a) Abra o arquivo ERB da action show (**app/views/restaurantes/show.html.erb**).

b) Adicione no final do ERB os seguintes links:

```
<%= link_to 'Editar', action: 'edit', id: @restaurante %>
<%= link_to 'Voltar', action: 'index' %>
```

9.24 Mostrando erros ao usuário

Desenvolvemos todas as actions e views necessárias para um CRUD de restaurante, porém não nos preocupamos de impedir que o usuário crie um restaurante inválido.

Felizmente, as validações que fizemos no modelo já impedem que restaurantes inválidos sejam salvos. Porém, se testarmos, veremos que ao tentarmos criar um restaurante inválido veremos uma página de erro.

Apesar de já estarmos protegidos contra o mau uso do sistema, nosso sistema responde de uma forma ofensiva, através de uma mensagem de erro.

Usuário ter a oportunidade de tentar novamente

Para resolver esse problema, mudaremos nossa action **create** para renderizar novamente o formulário caso um dado inválido esteja errado, dessa forma o usuário terá a oportunidade de corrigir os dados e enviá-los novamente.

Inicialmente, poderíamos pensar em utilizar o método **redirect_to**, dessa forma:

```
def create
  @restaurante = Restaurante.new params[:restaurante]

  if @restaurante.save
    redirect_to action: "show", id: @restaurante
  else
    redirect_to action: "new"
  end
end
```

Utilizando o **redirect_to** o usuário iria ver novamente o formulário caso digitasse dados inválidos, porém o formulário iria aparecer vazio e o usuário seria obrigado à digitar todas as informações novamente, ao invés de corrigir o erro. (Faça o teste)

Para resolvermos isso, iremos utilizar o método `render` para fazer com que a action **create** reaproveite a view **app/views/restaurantes/new.html.erb**. Dessa forma:

```
def create
  @restaurante = Restaurante.new params[:restaurante]

  if @restaurante.save
    redirect_to action: "show", id: @restaurante
  else
    render action: "new"
  end
end
```

A utilização do método render irá resultar em um formulário com os dados do restaurante, por que a view **new.html.erb** irá utilizar a variável `@restaurante` que foi criada na action **create**.

Avisar ao usuário os erros que foram cometidos

Apesar do usuário poder corrigir o erro, será difícil para ele enxergar sozinho que campos estão com erros.

Por isso, iremos mudar a view **new.html.erb** para mostrar os erros ao usuário. É importante lembrarmos o método **errors** que retorna um objeto que representa os erros do restaurante em questão. Iremos utilizar também o método **full_messages** que retorna uma array com as mensagens de erro.

Na nossa view **new.html.erb** iremos criar uma `<ul>` e iterar pela array de mensagens encaixando elas criando uma `<li>` para cada mensagem:

```
<ul>
<% @restaurante.errors.full_messages.each do |msg| %>
  <li><%= msg %></li>
<% end %>
</ul>
```

Feito isso, ficará fácil para o usuário tomar conhecimentos dos erros cometidos e consertá-los caso necessário.

9.25 Exercícios: Tratamento de erros na criação

1) O usuário não pode salvar um restaurante inválido, porém não há nada que avise o usuário. Vamos alterar nossa página de forma que sejam mostrados os erros do restaurante logo acima do formulário.

a) Abra a view do formulário de criação de um restaurante (**app/views/restaurantes/new.html.erb**).

b) Adicione a lista de erros logo acima do formulário:

```
<ul>
<% @restaurante.errors.full_messages.each do |msg| %>
  <li><%= msg %></li>
<% end %>
</ul>

<%= form_for @restaurante do |f| %>
<!-- resto do HTML -->
```

2) Vamos preparar o controller de restaurantes para quando a criação de um restaurante falhar nas validações. Nesse caso, ele deve renderizar o formulário de criação (**new.html.erb**) apartir do restaurante com erros:

a) Abra o seu controller de Restaurantes (**app/controllers/restaurantes_controllers.rb**)

b) Altere a action create para:

```
def create
  @restaurante = Restaurante.new(params[:restaurante])
  if @restaurante.save
    redirect_to action: "show", id: @restaurante
  else
    render action: "new"
  end
end
```

9.26 Mensagens de erro na atualização

Um problema parecido com o que experimentamos no momento da criação de um novo restaurante ocorrerá no processo de atualização de um restaurante. Para resolvê-lo, iremos alterar a action **update** para renderizar a view **edit.html.erb** caso o usuário tenha entrado com dados inválidos. Em seguida adicionaremos a listagem de erros na view **edit.html.erb** assim como fizemos na **new.html.erb**.

As alterações necessárias

Assim como na action **create**, também utilizaremos o método **render** na action **update**:

```
def update
  @restaurante = Restaurante.find params[:id]

  if @restaurante.update_attributes(params[:restaurante])
```

```
    redirect_to action: "show", id: @restaurante
  else
    render action: "edit"
  end
end
```

Para mostrar os erros ocorridos, assim como tivemos que implementar na view **new.html.erb**, precisaremos inserir a lista de erros na view **edit.html.erb**:

```
<ul>
<% @restaurante.errors.full_messages.each do |msg| %>
  <li><%= msg %></li>
<% end %>
</ul>

<%= form_for @restaurante do |f| %>
<!-- resto do HTML -->
```

9.27 Exercícios: Tratamento de erros na atualização

Assim como no formulário e action de criação, temos que preparar nossa aplicação para o caso de uma atualização de restaurante falhar em uma validação:

1) a) Abra a view do formulário de edição de um restaurante (**app/views/restaurantes/edit.html.erb**).

   b) Adicione a lista de erros logo acima do formulário:

```
<ul>
<% @restaurante.errors.full_messages.each do |msg| %>
  <li><%= msg %></li>
<% end %>
</ul>

<%= form_for @restaurante do |f| %>
<!-- resto do HTML -->
```

2) a) Abra o seu controller de Restaurantes (**app/controllers/restaurantes_controllers.rb**)

   b) Altere a action update para:

```
def update
  @restaurante = Restaurante.find(params[:id])

  if @restaurante.update_attributes(params[:restaurante])
    redirect_to action: "show", id: @restaurante
```

```
    else
      render action: "edit"
    end
  end
```

3) a) (Opcional) Quando o restaurante não tem erros, uma ul vazia é renderizada. Para evitar esse tipo de poluição, podemos envolver a **<ul>** com um if:

```
<% if @restaurante.errors.any? %>
  <!-- ul com erros aqui -->
<% end %>
```

b) (Opcional) Faça a mesma alteração do exercício acima na página **app/views/restaurantes/new.html.erb**.

9.28 Partial

Agora, suponha que eu queira exibir em cada página do restaurante um texto, por exemplo: "Controle de Restaurantes".

Poderíamos escrever esse texto manualmente, mas vamos aproveitar essa necessidade para conhecer um pouco sobre Partials.

Partials são fragmentos de *html.erb* que podem ser incluídas em uma view. Eles permitem que você reutilize sua lógica de visualização.

Para criar um Partial, basta incluir um arquivo no seu diretório de views (**app/views/restaurantes**) com o seguinte nome: `_meupartial`. Repare que Partials devem obrigatoriamente começar com _.

Para utilizar um Partial em uma view, basta acrescentar a seguinte linha no ponto que deseja fazer a inclusão:

```
render partial: "meupartial"
```

9.29 Exercícios: Reaproveitando fragmentos de ERB

1) Vamos criar um partial para reaproveitar o formulário que se repete nos templates **new.html.erb** e **edit.html.erb**:

a) Crie o arquivo: **app/views/restaurantes/_form.html.erb**

b) Coloque o seguinte conteúdo:

```
<ul>
<% @restaurante.errors.full_messages.each do |msg| %>
  <li><%= msg %></li>
```

```
<% end %>
</ul>

<%= form_for @restaurante do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= f.submit %>
<% end %>
```

2) a) Abra a view **new.html.erb**. (**app/views/restaurantes/new.html.erb**)

   b) Substitua a lista de erros e o formulário por:

```
<%= render partial: "form" %>
```

3) Repita o processo acima com a view **edit.html.erb**.

4) Teste novamente as actions **new** e **edit** acessando **http://localhost:3000/restaurantes**.

9.30 Respondendo em outros formatos como XML ou JSON

Um controlador pode ter diversos resultados. Em outras palavras, controladores podem responder de diversas maneiras, através do método `respond_to`:

```
class RestaurantesController < ApplicationController
  def index
    @restaurantes = Restaurante.all
    respond_to do |format|
      format.html {render "index"}
      format.xml {render xml: @restaurantes}
    end
  end
def
```

Dessa forma, conseguimos definir uma lógica de resposta diferente para cada formato.

Também é possível omitir o bloco de código:

```
class RestaurantesController < ApplicationController
  def index
    @restaurantes = Restaurante.all
    respond_to do |format|
      format.html
      format.xml {render xml: @restaurantes}
```

```
    end
  end
def
```

Por convenção, quando o bloco de código é omitido o Rails vai uma view que atenda ao seguinte padrão:

app/views/:controller/:action.:format.:handler

No caso de omitirmos para o formato HTML, o Rails irá renderizar a view: **app/views/restaurantes/index.html.erb**

9.31 Para saber mais: Outros handlers

O Rails já vem com suporte a outros *handlers* para geração de views. Além do ERB, podemos também usar o **Builder**.

O **Builder** é adequado quando a view a ser gerada é um arquivo XML, já que permite a criação de um xml usando sintaxe Ruby. Veja um exemplo:

```
# app/views/authors/show.xml.builder
xml.author do
  xml.name('Alexander Pope')
end
```

O xml resultante é:

```
<author>
  <name>Alexander Pope</name>
</author>
```

Outra alternativa muito popular para a geração das views é o **HAML**:

http://haml.hamptoncatlin.com

```
#content
  .left.column
    %h2 Welcome to our site!
    %p= print_information
  .right.column= render partial: "sidebar"
```

E o equivalente com ERB:

```
<div id='content'>
  <div class='left column'>
    <h2>Welcome to our site!</h2>
    <p>
      <%= print_information %>
    </p>
  </div>
  <div class="right column">
    <%= render partial: "sidebar" %>
  </div>
</div>
```

9.32 Exercícios: Disponibilizando restaurantes como XML e JSON

1) Vamos deixar explícito que a action **index** do controller de restaurantes responder com html:

   a) Abra o controller de restaurantes. (**app/controllers/restaurantes_controller.rb**)

   b) Altere a action **index** para:

```
def index
  @restaurantes = Restaurante.order :nome

  respond_to do |format|
    format.html
  end
end
```

2) Agora que temos a forma explícita de declarar formatos de resposta, será simples capacitar a action **index** para responder com **XML** e **JSON**:

   a) Ainda no controller de restaurantes. (**app/controllers/restaurantes_controller.rb**)

   b) Altere a action **index** para:

```
def index
  @restaurantes = Restaurante.order :nome

  respond_to do |format|
    format.html
    format.xml {render xml: @restaurantes}
    format.json {render json: @restaurantes}
  end
end
```

3) Teste os novos formatos acessando as urls:

   a) http://localhost:3000/restaurantes.xml

   b) http://localhost:3000/restaurantes.json

9.33 Exercícios Opcionais: Outras actions respondendo XML e JSON

Já ensinamos o controller de restaurantes a responder um conjunto de restaurantes. Vamos implementar para que ele conseguir responder com um JSON ou XML representando apenas um restaurante.

1) a) Abra o controller de restaurantes. (**app/controllers/restaurantes_controller.rb**)

   b) Altere a action **show** para:

```
def show
  @restaurante = Restaurante.find params[:id]

  respond_to do |format|
    format.html
    format.json {render json: @restaurante}
    format.xml {render xml: @restaurante}
  end
end
```

2) Para testar, procure pelo `id` de algum restaurante e acesse: **http://localhost:3000/restaurantes/<id>.json**.

9.34 Filtros

O módulo `ActionController::Filters` define formas de executar código antes e depois de todas as actions.

Para executar código antes das actions:

```
class ClientesController < ApplicationController
  before_filter :verifica_login

  private
  def verifica_login
    redirect_to controller: 'login' unless usuario_logado?
  end
end
```

De forma análoga, podemos executar código no fim do tratamento da requisição:

```
class ClientesController < ApplicationController
  after_filter :avisa_termino

  private
  def avisa_termino
    logger.info "Action #{params[:action]} terminada"
  end
end
```

Por fim, o mais poderoso de todos, que permite execução de código tanto antes, quanto depois da action a ser executada:

```
class ClientesController < ApplicationController
  around_filter :envolvendo_actions

  private
  def envolvendo_actions
    logger.info "Antes de #{params[:action]}: #{Time.now}"
    yield
    logger.info "Depois de #{params[:action]}: #{Time.now}"
  end
end
```

Os filtros podem também ser definidos diretamente na declaração, através de blocos:

```
class ClientesController < ApplicationController
  around_filter do |controller, action|
    logger.info "#{controller} antes: #{Time.now}"
    action.call
    logger.info "#{controller} depois: #{Time.now}"
  end
end
```

Caso não seja necessário aplicar os filtros a todas as actions, é possível usar as opções `:except` e `:only`:

```
class ClientesController < ApplicationController
  before_filter :verifica_login, :only => [:create, :update]

  # ...
end
```

Logger

As configurações do log podem ser feitas através do arquivo `config/environment.rb`, ou especificamente para cada environment nos arquivos da pasta `config/environments`. Entre as configurações que podem ser customizadas, estão qual nível de log deve ser exibido e para onde vai o log (stdout, arquivos, email, ...).

```
Rails::Initializer.run do |config|
  # ...
  config.log_level = :debug
  config.log_path = 'log/debug.log'
  # ...
end
```

Mais detalhes sobre a customização do log podem ser encontrados no wiki oficial do Rails: http://wiki.rubyonrails.org/rails/show/HowtoConfigureLogging

Capítulo 10

Completando o Sistema

"O êxito parece doce a quem não o alcança"
– Dickinson, Emily

10.1 Um pouco mais sobre o Scaffold

Vimos que quando usamos o gerador `scaffold` o Rails cria os arquivos necessários em todas as camadas. **Controller**, **Model**, **Views** e até mesmo arquivos de testes e a **Migration**. Para concluir nossos projeto precisamos criar apenas **Controller + Views** pois tanto o modelo quanto as migrations já estão prontos. Para isso vamos usar o mesmo gerador scaffold mas vamos passar os parâmetros: `--migration=false` para ele ignorar a criação da migration e o parâmetro `-s` que é a abreviação para `--skip` que faz com que o rails "pule" os arquivos já existentes.

Para concluir os modelos que já começamos vamos executar o gerador da seguinte maneira: **rails generate scaffold cliente nome:string idade:integer --migration=false -s**

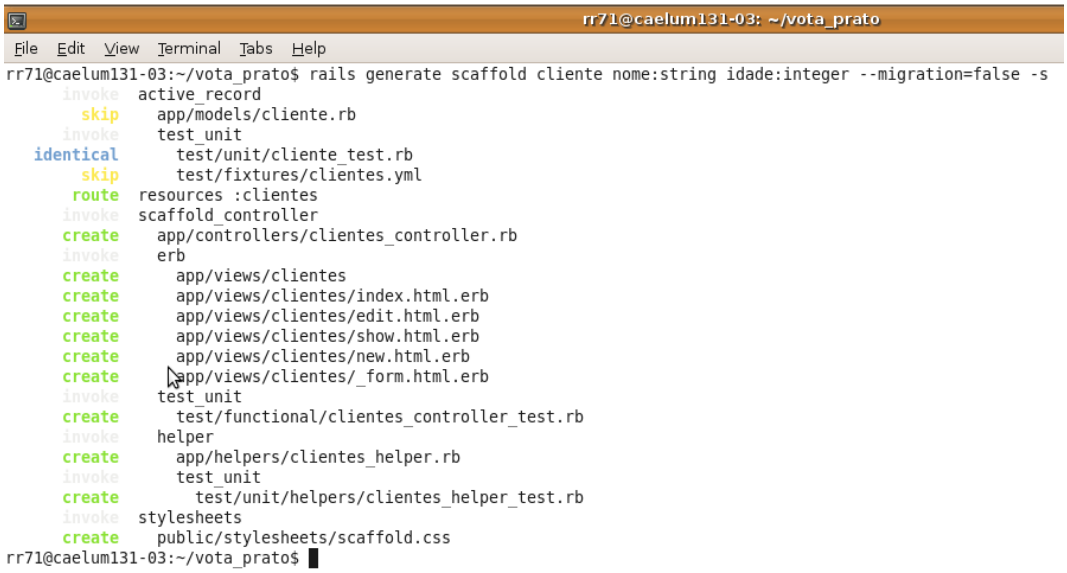
> **Outros parâmetros nos geradores**
>
> Para conhecer todas as opções de parâmetros que um gerador pode receber tente executá-lo passando o parâmetro `-h` Ex. **rails generate scaffold -h**

10.2 Exercícios: Completando nosso domínio

Vamos gerar os outros controllers e views usando o scaffold:

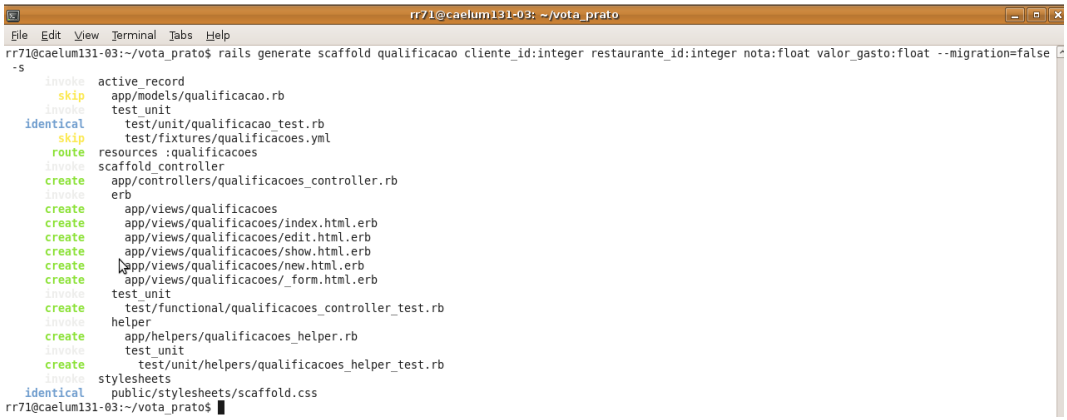
1) Primeiro vamos gerar o scaffold para **cliente**, no terminal execute:

```
$ rails generate scaffold cliente nome idade:integer --migration=false -s
```

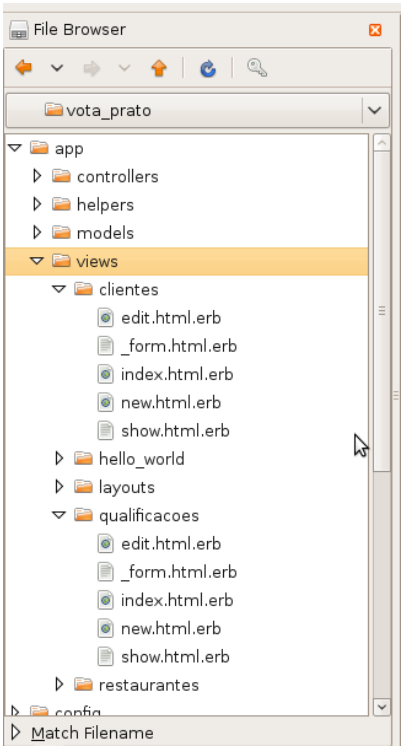


2) Agora vamos gerar o scaffold de **qualificacao**, execute:

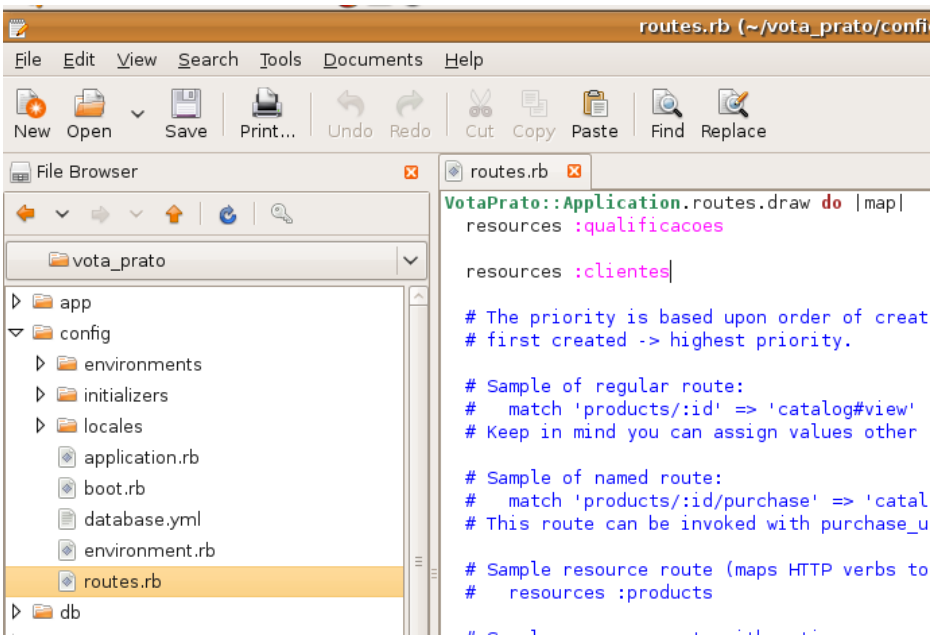
```
$ rails generate scaffold qualificacao cliente_id:integer
    restaurante_id:integer nota:float valor_gasto:float
    --migration=false -s
```



3) a) Olhe as views criadas (**app/views/clientes** e **app/views/qualificacoes**)



b) Olhe as rotas criadas (**config/routes.rb**)



c) Abra os arquivos **app/views/clientes/index.html.erb** e **app/views/restaurantes/index.html.erb** e apague as linhas que chamam a action `destroy` Lembre-se de que não queremos inconsistências na nossa tabela de qualificações

d) Reinicie o servidor

e) Teste: **http://localhost:3000/clientes** e **http://localhost:3000/qualificacoes**

Note que precisamos utilizar a opção "--migration=false" no comando `scaffold`, além de informar manualmente os atributos utilizados em nossas migrations. Isso foi necessário, pois já tínhamos um migration pronto, e queríamos que o Rails gerasse os formulários das views para nós, e para isso ele precisaria conhecer os atributos que queríamos utilizar.

> **CSS SCAFFOLD**
>
> O comando `scaffold`, quando executado, gera um css mais bonito para nossa aplicação. Se quiser utilizá-lo, edite nosso layout (**app/views/layouts/application.html.erb**) e adicione a seguinte linha logo abaixo da tag `<title>`:
>
> <%= stylesheet_link_tag 'scaffold' %>

10.3 Selecionando Clientes e Restaurante no form de Qualificações

Você já deve ter reparado que nossa view de adição e edição de qualificações está um tanto quanto estranha: precisamos digitar os IDs do cliente e do restaurante manualmente.

Para corrigir isso, podemos utilizar o **FormHelper** `select`, inserindo o seguinte código nas nossas views de adição e edição de qualificações:

```
<%= select('qualificacao', 'cliente_id',
                           Cliente.order(:nome)
                           {|p| [ p.nome, p.id]}) %>
```

em substituição ao:

```
<%= f.number_field :cliente_id %>
```

Mas existe um outro **FormHelper** mais elegante, que produz o mesmo efeito, o `collection_select`:

```
<%= collection_select(:qualificacao, :cliente_id,
    Cliente.order(:nome),
    :id, :nome, {:prompt => true}) %>
```

Como estamos dentro de um form_for, podemos usar do fato de que o formulário sabe qual o nosso Active-Record, e com isso fazer apenas:

```
<%= f.collection_select(:cliente_id,
    Cliente.order(:nome),
    :id, :nome, {:prompt => true}) %>
```

10.4 Exercícios: Formulário com collection_select

1) Vamos utilizar o **FormHelper** `collection_select` para exibirmos o nome dos clientes e restaurantes nas nossas views da qualificação:

   a) Abra o arquivo **app/views/qualificacoes/_form.html.erb**

   b) Troque a linha:

   ```
   <%= f.number_field :cliente_id %>
   ```

   por:

   ```
   <%= f.collection_select(:cliente_id, Cliente.order(:nome),
                                        :id, :nome, {:prompt => true}) %>
   ```

   c) Troque a linha:

   ```
   <%= f.number_field :restaurante_id %>
   ```

   por:

   ```
   <%= f.collection_select(:restaurante_id, Restaurante.order(:nome),
                                           :id, :nome, {:prompt => true}) %>
   ```

   d) Teste: **http://localhost:3000/qualificacoes**

10.5 Exercícios Opcionais: Refatorando para respeitarmos o MVC

A forma como implementamos o seletor de restaurante e cliente não é a mais adequada pois ela fere o modelo arquitetural que o rails se baseia, o MVC. Isso ocorre, pois estamos invocando `Cliente.order(:nome)` dentro da view, ou seja, a view está se comunicando com o modelo.

1) Vamos arrumar isso, criando as variáveis de instância `@restaurantes` e `@clientes` nas actions **new** e **edit** do **QualificacoesController**:

   a) Abra o **QualificacoesController**. (**app/controllers/qualificacoes_controller.rb**)

   b) Crie as variáveis de instância nas primeiras linhas da action **new**:

```
def new
    @clientes = Cliente.order :nome
    @restaurantes = Restaurante.order :nome

    # resto do código
end
```

c) Crie as variáveis de instância nas primeiras linhas da action **edit**:

```
def edit
    @clientes = Cliente.order :nome
    @restaurantes = Restaurante.order :nome

    # resto do código
end
```

2) Agora podemos usar as variáveis de instância no nosso partial **form**.

a) Abra o partial **form** (**app/views/qualificacoes/_form.html.erb**)

b) Substitua as antigas chamadas do `select_collection` por:

```
f.collection_select(:restaurante_id, @restaurantes,
                                :id, :nome, {:prompt => true})
```

e

```
f.collection_select(:restaurante_id, @clientes,
                                :id, :nome, {:prompt => true})
```

3) Observe que temos código duplicado no nosso **QualificacoesController**, vamos refatorar extraindo a criação das variáveis de instância para um método privado,

a) Abra o controller de qualificacoes (**app/controllers/qualificacoes_controller.rb**).

b) Ao final da classe, crie um método privado que cria as variáveis de instância:

```
class QualificacoesController < ApplicationController
  # todas as actions

private
  def preparar_form
              @clientes = Cliente.order :nome
              @restaurantes = Restaurante.order :nome
          end
end
```

c) Ainda no controller de qualificacoes, vamos usar o `preparar_form` na action **new**. Dentro da action **new**, substitua:

```
@clientes = Cliente.order :nome
@restaurantes = Restaurante.order :nome
```

por:

```
preparar_form
```

d) Repita o exercício acima na action **edit**.

4) As actions **new** e **edit** estão funcionando perfeitamente. Vamos analisar a action **create**. Observe que caso dê erro de validação, ela também irá renderizar a view **new.html.erb**. Logo, é necessário criarmos as variáveis de instância `@restaurantes` e `@clientes` também na action **create**.

a) Abra o controller de qualificacoes (**app/controllers/qualificacoes_controller.rb**).

b) Altere a action **create** para chamar o **preparar_form**:

```
def create
  @qualificacao = Qualificacao.new(params[:qualificacao])

  respond_to do |format|
    if @qualificacao.save
      format.html { redirect_to @qualificacao,
                    notice: 'Qualificacao was successfully created.' }
      format.json { render json: @qualificacao,
                    status: :created, location: @qualificacao }
    else
      preparar_form
      format.html { render action: "new" }
      format.json { render json: @qualificacao.errors, status: :unprocessable_entity }
    end
  end
end
```

5) O mesmo problema da action **create** ocorre na action **update**, portanto vamos alterá-la.

a) Abra o controller de qualificacoes (**app/controllers/qualificacoes_controller.rb**).

b) Altere a action **update** para chamar o **preparar_form**:

```
def update
  @qualificacao = Qualificacao.find(params[:id])

  respond_to do |format|
    if @qualificacao.update_attributes(params[:qualificacao])
      format.html { redirect_to @qualificacao,
                    notice: 'Qualificacao was successfully updated.' }
      format.json { head :no_content }
    else
```

```
        preparar_form
        format.html { render action: "edit" }
        format.json { render json: @qualificacao.errors,
                              status: :unprocessable_entity }
      end
    end
  end
```

6) Inicie o servidor (`rails server`) e tente criar uma qualificação inválida para testar nossa correção.

10.6 Exercícios Opcionais: Exibindo nomes ao invés de números

1) Agora vamos exibir o nome dos restaurantes e clientes nas views **index** e **show** de qualificações:

   a) Abra o arquivo **app/views/qualificacoes/show.html.erb**

   b) Troque a linha:

   ```
   <%= @qualificacao.cliente_id %>
   ```

   por:

   ```
   <%= @qualificacao.cliente.nome %>
   ```

   c) Troque a linha:

   ```
   <%= @qualificacao.restaurante_id %>
   ```

   por:

   ```
   <%= @qualificacao.restaurante.nome %>
   ```

   d) Abra o arquivo **app/views/qualificacoes/index.html.erb**

   e) Troque as linhas:

   ```
   <td><%= qualificacao.cliente_id %></td>
   <td><%= qualificacao.restaurante_id %></td>
   ```

   por:

   ```
   <td><%= qualificacao.cliente.nome %></td>
   <td><%= qualificacao.restaurante.nome %></td>
   ```

   f) Teste: **http://localhost:3000/qualificacoes**

2) Por fim, vamos utilizar o FormHelper hidden_field para permitir a qualificação de um restaurante a partir da view **show** de um cliente ou de um restaurante. No entanto, ao fazer isso, queremos que não seja necessário a escolha de cliente ou restaurante. Para isso:

a) Abra o arquivo **app/views/qualificacoes/_form.html.erb**

b) Troque as linhas

```
<p>
  <%= f.label :cliente_id %><br />
  <%= f.collection_select(:cliente_id,
      Cliente.order('nome'),
      :id, :nome, {:prompt => true}) %>
</p>
```

por:

```
<% if @qualificacao.cliente %>
  <%= f.hidden_field 'cliente_id' %>
<% else %>
  <p><%= f.label :cliente_id %><br />
<%= f.collection_select(:cliente_id, Cliente.order('nome'),
                                    :id, :nome, {:prompt => true}) %></p>
<% end %>
```

c) Troque as linhas

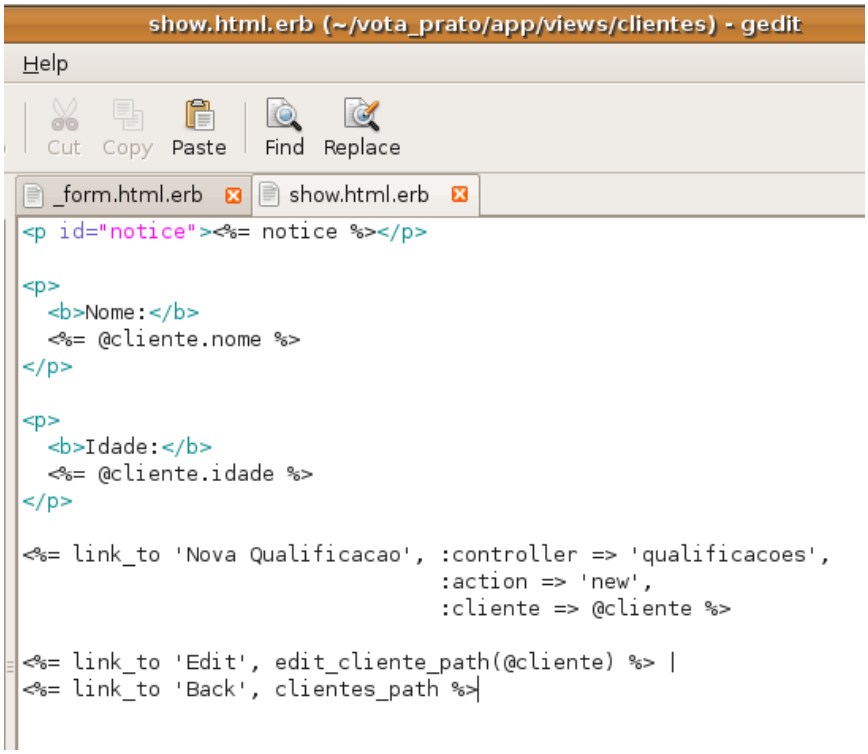
```
<p>
  <%= f.label :restaurante_id %><br />
  <%= f.collection_select(:restaurante_id, Restaurante.order('nome'),
                                          :id, :nome, {:prompt => true}) %>
</p>
```

por:

```
<% if @qualificacao.restaurante %>
  <%= f.hidden_field 'restaurante_id' %>
<% else %>
  <p><%= f.label :restaurante_id %><br />
  <%= f.collection_select(:restaurante_id, Restaurante.order('nome'),
                                          :id, :nome, {:prompt => true}) %></p>
<% end %>
```

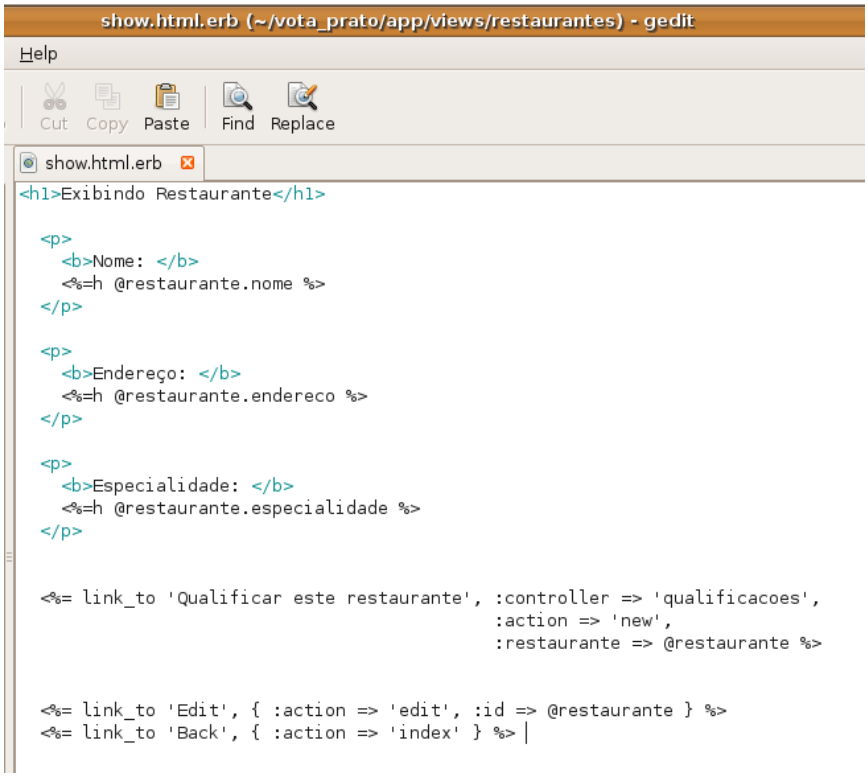
d) Adicione a seguinte linha na view **show** do cliente (**app/views/clientes/show.html.erb**):

```
<%= link_to "Nova qualificação", :controller => "qualificacoes",
                                 :action => "new",
                                 :cliente => @cliente %>
```



e) Adicione a seguinte linha na view **show** do restaurante (**app/views/restaurantes/show.html.erb**):

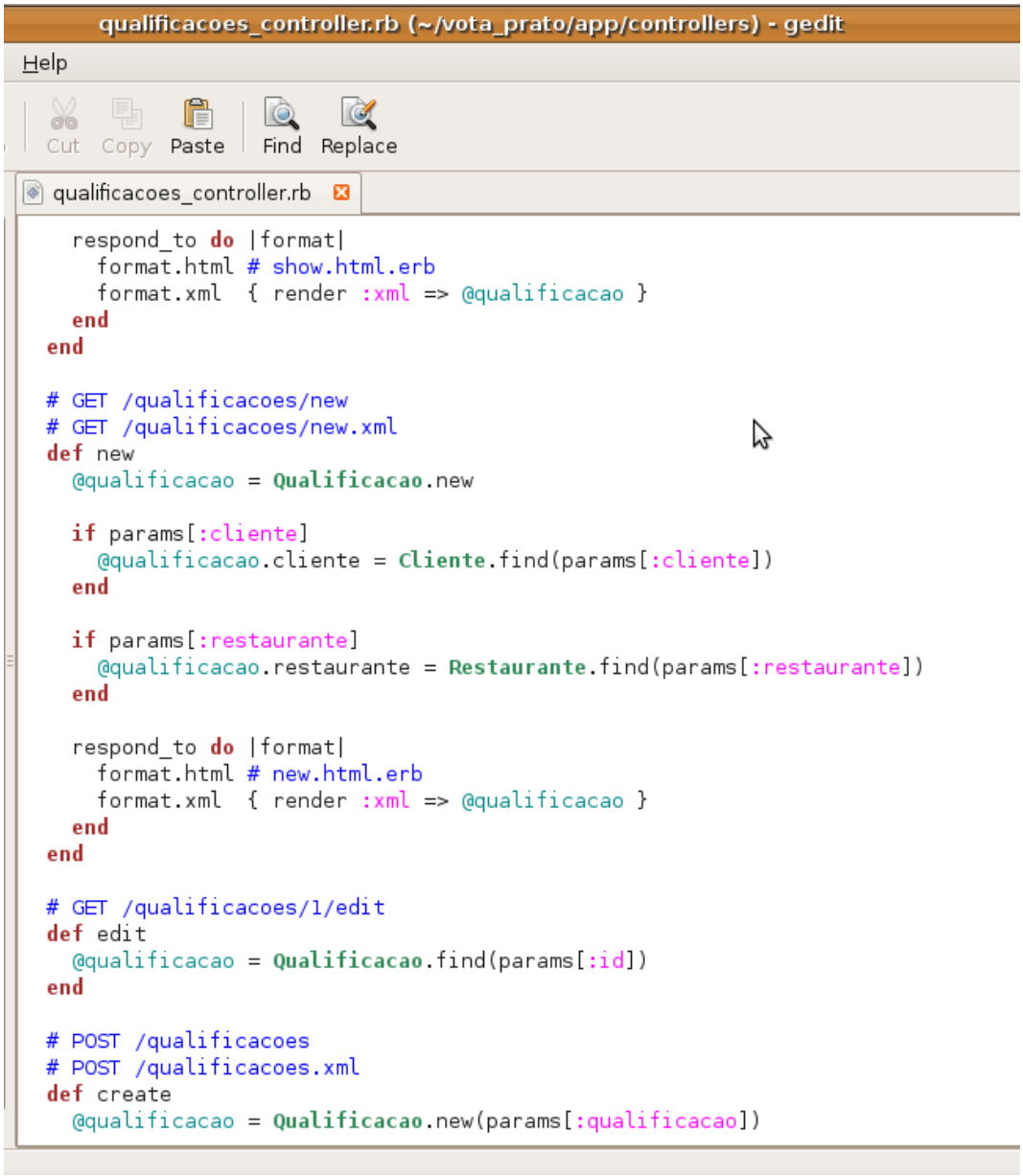
```
<%= link_to "Qualificar este restaurante",  :controller => "qualificacoes",
                                            :action => "new",
                                            :restaurante => @restaurante %>
```



f) Por fim, precisamos fazer com que o controlador da action `new` das qualificações receba os parâmetros para preenchimento automático. Abra o controller **app/controllers/qualificacoes_controller.rb**

g) Adicione as seguintes linhas à nossa action `new`:

```
if params[:cliente]
  @qualificacao.cliente = Cliente.find(params[:cliente])
end
if params[:restaurante]
  @qualificacao.restaurante = Restaurante.find(params[:restaurante])
end
```



h) Teste: **http://localhost:3000/clientes**, entre na página Show de um cliente e faça uma nova qualificação.

10.7 Mais sobre os controllers

Podemos notar que nossas actions, por exemplo a `index`, fica muito parecida com a action `index` de outros controladores, mudando apenas o nome do modelo em questão.

```
#qualificacoes_controller
def index
  @qualificacoes = Qualificacao.all

  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @qualificacoes }
  end
end

#clientes_controller
def index
  @clientes = Cliente.all

  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @clientes }
  end
end
```

Normalmente precisamos fazer exatamente a mesma ação para formatos iguais e por isso acabamos repetindo o mesmo bloco de `respond_to` nas actions. Para solucionar esse problema, no rails 3 acrescentaram o método `respond_to` **nos controllers** e o método `respond_with` **nas actions**. Veja o exemplo:

```
class ClientesController < ApplicationController

  respond_to :html, :xml

  # GET /clientes
  # GET /clientes.xml
  def index
    @clientes = Cliente.all

    respond_with @clientes
  end

  # GET /clientes/1
  # GET /clientes/1.xml
  def show
```

```
    @cliente = Cliente.find(params[:id])

    respond_with @cliente
  end
  ...
end
```

Dessa forma estamos dizendo para o rails que esse controller irá responder para os formatos html e xml, dentro da action basta eu dizer qual objeto é pra ser usado. No caso da action **index**, se a requisição pedir o formato html, o rails simplesmente vai enviar a chamada para o arquivo `views/clientes/index.html.erb` e a variável @clientes estará disponível lá. Se a requisição pedir o formato xml, o rails fará exatamente o mesmo que estava no bloco de respond_to que o scaffold criou, vai renderizar a variável @clientes em xml. O bloco de código acima é equivalente ao gerado pelo scaffold:

```
class ClientesController < ApplicationController
  # GET /clientes
  # GET /clientes.xml
  def index
    @clientes = Cliente.all

    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @clientes }
    end
  end

  # GET /clientes/1
  # GET /clientes/1.xml
  def show
    @cliente = Cliente.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @cliente }
    end
  end
   ...
end
```

Veja na imagem como ficaria o **ClientesController** usando essa outra maneira de configurar os controllers.

clientes_controller.rb

```
class ClientesController < ApplicationController

  respond_to :html, :xml

  # GET /clientes
  # GET /clientes.xml
  def index
    @clientes = Cliente.all

    respond_with @clientes
  end

  # GET /clientes/1
  # GET /clientes/1.xml
  def show
    @cliente = Cliente.find(params[:id])

    respond_with @cliente
  end

  # GET /clientes/new
  # GET /clientes/new.xml
  def new
    @cliente = Cliente.new

    respond_with @cliente
  end

  # GET /clientes/1/edit
  def edit
    @cliente = Cliente.find(params[:id])
  end

  # POST /clientes
```

Capítulo 11

Calculations

"Ao examinarmos os erros de um homem conhecemos o seu caráter"
– Chamfort, Sébastien Roch

Nesse capítulo, você aprenderá a utilizar campos para calcular fórmulas como, por exemplo, a média de um campo.

11.1 Métodos

Uma vez que existem os campos valor gasto e nota, seria interessante disponibilizar para os visitantes do site a média de cada um desses campos para determinado restaurante.

Em Rails esse recurso é chamado **calculations**, métodos dos nossos modelos que fazem operações mais comuns com campos numéricos como, por exemplo:

- average(column_name, options = {}) - média
- maximum(column_name, options = {}) - maior valor
- minimum(column_name, options = {}) - menor valor
- sum(column_name, options = {}) - soma
- count(*args) - número de entradas

11.2 Média

Supondo que o cliente pediu para adicionar a nota média de um restaurante na tela com as informações do mesmo (**show**). Basta adicionar uma chamada ao método `average` das qualificações do nosso restaurante:

```
<b>Nota média: </b><%= @restaurante.qualificacoes.average(:nota) %><br/>
```

Podemos mostrar também o número total de qualificações que determinado restaurante possui:

```
<b>Qualificações: </b><%= @restaurante.qualificacoes.count %><br/>
```

E, por último, fica fácil adicionar o valor médio gasto pelos clientes que visitam tal restaurante:

```
<b>Preço médio: </b><%= @restaurante.qualificacoes.average(:valor_gasto) %><br/>
```

11.3 Exercícios

1) Altere a view **show** de restaurante para mostrar sua nota média, quantas qualificações possui e preço médio:

   a) Insira as seguintes linhas em **app/views/restaurantes/show.html.erb**:

```
<b>Nota média: </b>
      <%= @restaurante.qualificacoes.average(:nota) %><br/>
<b>Qualificações: </b>
      <%= @restaurante.qualificacoes.count %><br/>
<b>Preço médio: </b>
      <%= @restaurante.qualificacoes.average(:valor_gasto) %><br/><br/>
```

show.html.erb (~/vota_prato/app/views/restaurantes) - gedit

Help

Cut Copy Paste Find Replace

show.html.erb

```
<h1>Exibindo Restaurante</h1>

  <p>
    <b>Nome: </b>
    <%=h @restaurante.nome %>
  </p>

  <p>
    <b>Endereço: </b>
    <%=h @restaurante.endereco %>
  </p>

  <p>
    <b>Especialidade: </b>
    <%=h @restaurante.especialidade %>
  </p>

<b>Nota media: <%= @restaurante.qualificacoes.average(:nota)%></b>
<b>Qualificacões: <%= @restaurante.qualificacoes.count%></b>
<b>Preco Medio: <%= @restaurante.qualificacoes.average(:valor_gasto)%></b>


  <%= link_to 'Qualificar este restaurante', :controller => 'qualificacoes',
                                             :action => 'new',
                                             :restaurante => @restaurante %>


  <%= link_to 'Edit', { :action => 'edit', :id => @restaurante } %>
  <%= link_to 'Back', { :action => 'index' } %>
```

b) Entre no link **http://localhost:3000/restaurantes** e escolha um restaurante para ver suas estatísticas.

Capítulo 12

Associações Polimórficas

"Os negócios são o dinheiro dos outros"
– Alexandre Dumas

Nesse capítulo você verá como criar uma relação *muitos-para-muitos* para mais de um tipo de modelo.

12.1 Nosso problema

O cliente pede para a equipe de desenvolvedores criar uma funcionalidade que permita aos visitantes deixar comentários sobre suas visitas aos restaurantes.

Para complicar a vida do programador, o cliente pede para permitir comentários também em qualificações, permitindo aos usuários do site justificar a nota que deram.

Esse problema poderia ser resolvido de diversas maneiras sendo que trabalharemos em cima de um modelo para representar um comentário, relacionado com restaurantes e qualificações, aproveitando para mostrar como realizar tal tipo de relacionamento.

Seria simples se pudéssemos criar mais uma tabela com o comentário em si e o `id` da entidade relacionada. O problema surge no momento de diferenciar um comentário sobre qualificação de um sobre restaurante.

Para diferenciar os comentários de restaurantes e qualificações, podemos usar um atributo de nome "tipo".

Em Ruby podemos criar apelidos para um ou mais modelos, algo similar a diversas classes implementarem determinada interface (sem métodos) em java. Podemos chamar nossos modelos `Restaurante` e `Qualificacao` como comentáveis, por exemplo.

Um exemplo dessa estrutura em Java é o caso de `Serializable` - interface que não obriga a implementação de nenhum método mas serve para marcar classes como serializáveis, sendo que diversas classes da api padrão do Java implementam a primeira.

No caso do Ruby, começamos criando um modelo chamado `Comentario`.

12.2 Alterando o banco de dados

O conteúdo do script de migração criará as colunas "comentário", "id de quem tem o comentário", e o "tipo".

Nos campos `id` e `tipo`, colocamos o nome da coluna com o apelido seguido de `_id` e `_type`, respectivamente, notificando o Ruby que ele deve buscar tais dados daquilo que é "comentavel".

Note que no português a palavra "comentavel" soa estranho e parece esquisito trabalhar com ela, mas para seguir o padrão definido no inglês em diversas linguagens, tal apelido indica o que os modelos são capazes de fazer e, no caso, eles são "comentáveis".

O script deve então criar três colunas, sem nada de novo comparado com o que vimos até agora:

```
rails generate scaffold comentario conteudo:text \
        comentavel_id:integer comentavel_type
```

Caso seja necessário, podemos ainda adicionar índices físicos nas colunas do relacionamento, deixando a migration criada como a seguir:

```
class CreateComentarios < ActiveRecord::Migration
  def change
    create_table :comentarios do |t|
      t.text :conteudo
      t.integer :comentavel_id
      t.string :comentavel_type

      t.timestamps
    end

    add_index :comentarios, :comentavel_type
    add_index :comentarios, :comentavel_id
  end
end
```

Para trabalhar nos modelos, precisamos antes gerar a nova tabela necessária:

```
$ rake db:migrate
```

O modelo `Comentario` (app/models/comentario.rb) deve poder ser associado a qualquer objeto do grupo de modelos comentáveis. Qualquer objeto poderá fazer o papel de `comentavel`, por isso dizemos que a associação é polimórfica:

```
class Comentario < ActiveRecord::Base
  attr_accessible :conteudo, :comentavel_id, :comentavel_type
  belongs_to :comentavel, polymorphic: true
end
```

A instrução `:polymorphic` indica a não existência de um modelo com o nome `:comentavel`.

Falta agora comentar que uma qualificação e um restaurante terão diversos comentários, fazendo o papel de algo `comentavel`. Para isso usaremos o relacionamento has_many:

```
class Qualificacao < ActiveRecord::Base
  # ...

  belongs_to :cliente
  belongs_to :restaurante

  has_many :comentarios, as: comentavel

  # ...
end
```

E o Restaurante:

```
class Restaurante < ActiveRecord::Base
  # ...

  has_many :qualificacoes
  has_many :comentarios, as: comentavel

  # ...
end
```

A tradução do texto pode ser quase literal: o modelo **TEM MUITOS** comentários **COMO** comentável.

12.3 Exercícios: Criando modelo de comentário

1) Vamos criar o modelo do nosso comentário e fazer a migração para o banco de dados:

   a) Vá ao Terminal

   b) Digite:

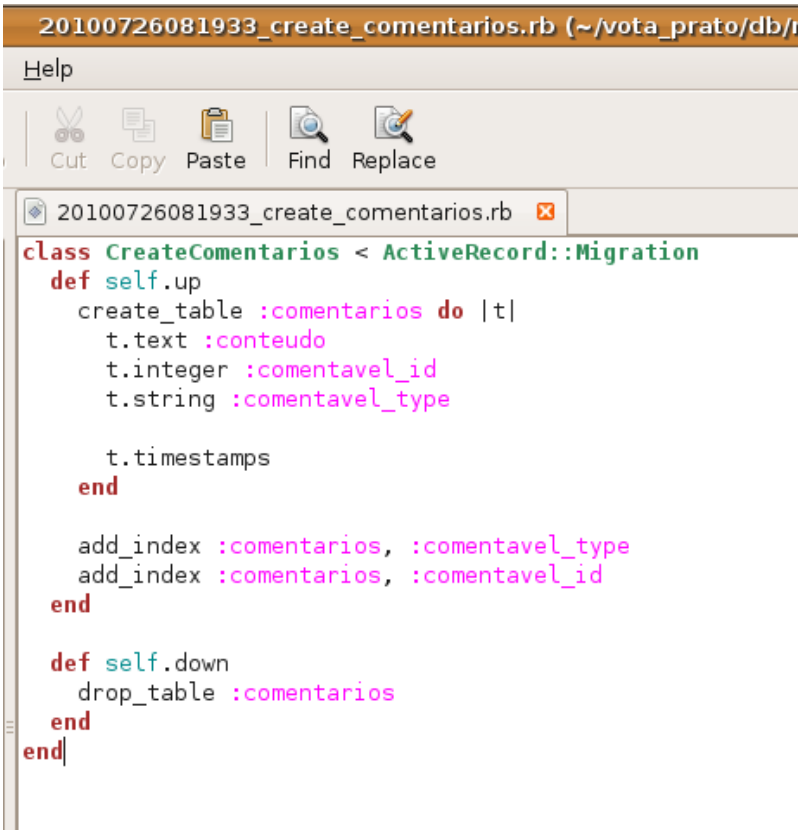
```
$ rails generate scaffold comentario conteudo:text
    comentavel_id:integer comentavel_type
```

```
rr71@caelum131-03: ~/vota_prato
File  Edit  View  Terminal  Tabs  Help
rr71@caelum131-03:~/vota_prato$ rails generate scaffold comentario conteudo:text comentavel_id:integer comentavel_type:string
      invoke  active_record
      create    db/migrate/20100726081933_create_comentarios.rb
      create    app/models/comentario.rb
      invoke    test_unit
      create      test/unit/comentario_test.rb
      create      test/fixtures/comentarios.yml
       route  resources :comentarios
      invoke  scaffold_controller
      create    app/controllers/comentarios_controller.rb
      invoke    erb
      create      app/views/comentarios
      create      app/views/comentarios/index.html.erb
      create      app/views/comentarios/edit.html.erb
      create      app/views/comentarios/show.html.erb
      create      app/views/comentarios/new.html.erb
      create      app/views/comentarios/_form.html.erb
      invoke    test_unit
      create      test/functional/comentarios_controller_test.rb
      invoke    helper
      create      app/helpers/comentarios_helper.rb
      invoke      test_unit
      create        test/unit/helpers/comentarios_helper_test.rb
      invoke  stylesheets
   identical    public/stylesheets/scaffold.css
rr71@caelum131-03:~/vota_prato$
```

c) Vamos inserir alguns índices físicos. Abra o arquivo **db/migrate/<timestamp>_create_comentarios.rb**

d) Insira as seguintes linhas:

```
add_index :comentarios, :comentavel_type
add_index :comentarios, :comentavel_id
```



e) Volte ao Terminal e rode as migrações com o comando:

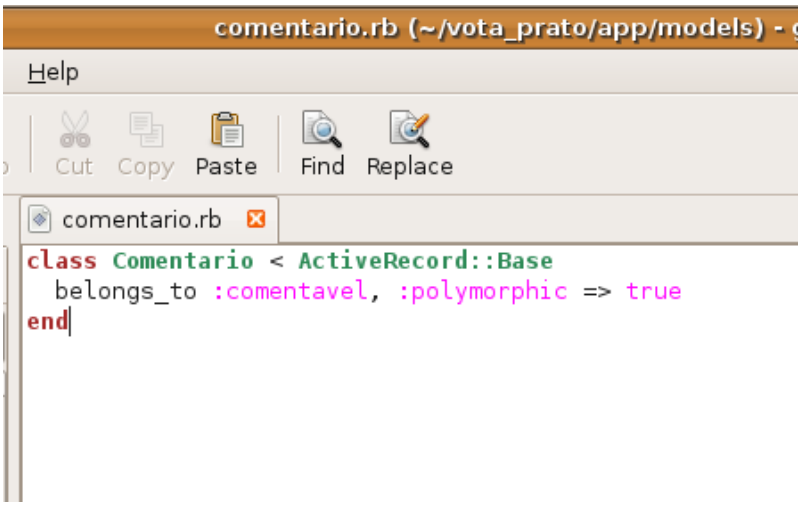
```
$ rake db:migrate
```

2) Vamos modificar nossos modelos:

   a) Abra o arquivo **app/models/comentario.rb**

   b) Adicione a seguinte linha:

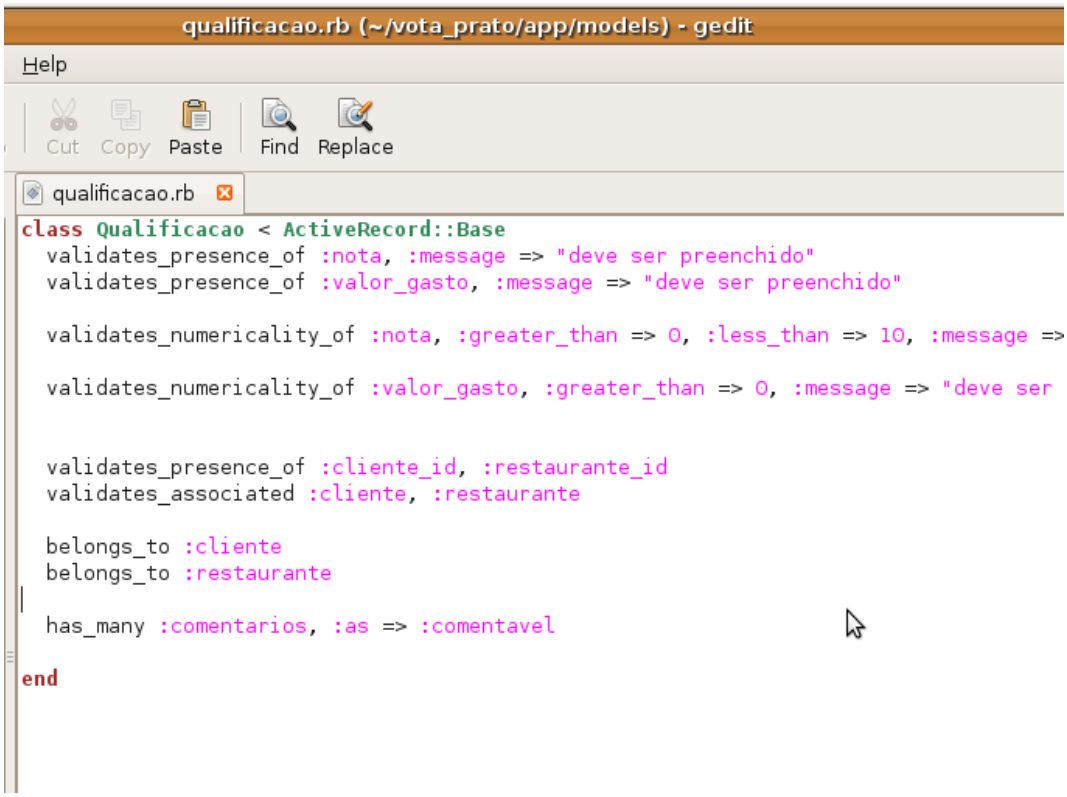
```
belongs_to :comentavel, polymorphic: true
```



   c) Abra o arquivo **app/models/qualificacao.rb**

   d) Adicione a seguinte linha:

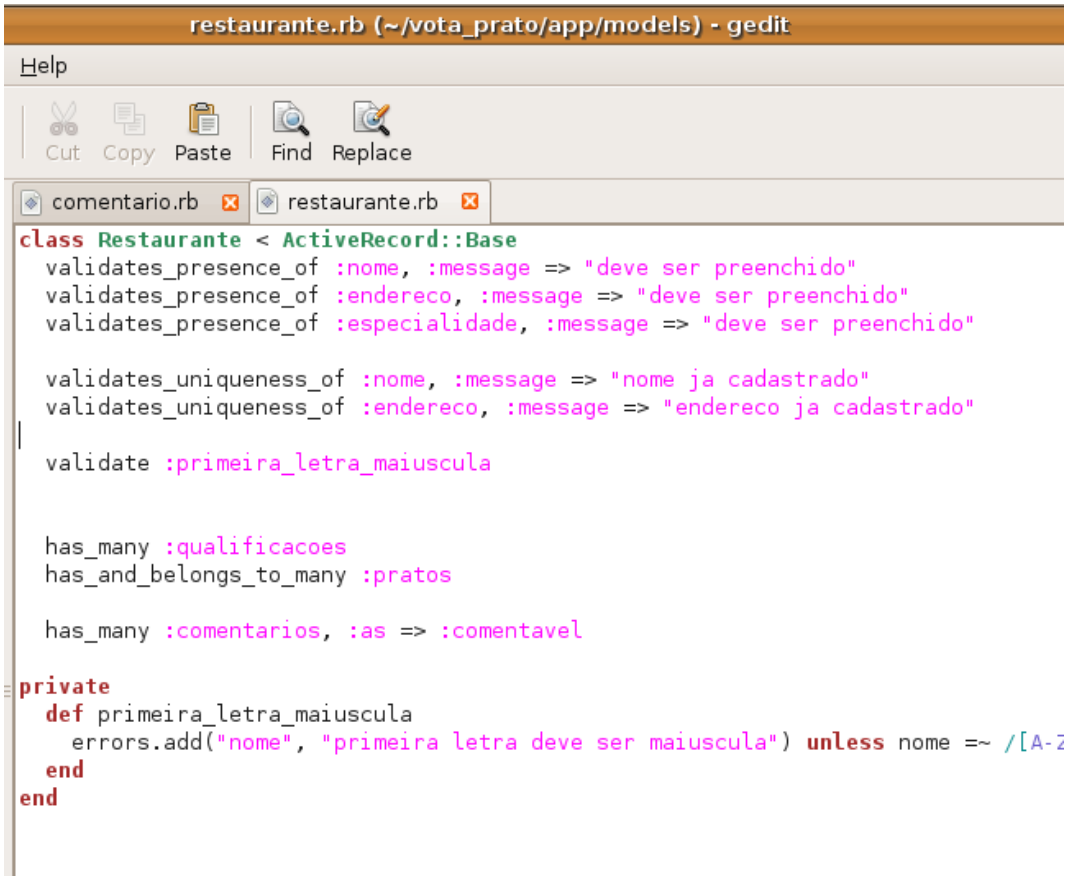
```
has_many :comentarios, as: :comentavel
```



e) Abra o arquivo **app/models/restaurante.rb**

f) Adicione a seguinte linha:

```
has_many :comentarios, as: :comentavel
```



3) Para o próximo capítulo, vamos precisar que o nosso sistema já inclua alguns comentários. Para criá-los, você pode usar o `rails console` ou ir em **http://localhost:3000/comentarios** e adicionar um comentário qualquer para o "Comentavel" 1, por exemplo, e o tipo "Restaurante". Isso criará um comentário para o restaurante de ID 1.

Capítulo 13

Mais sobre views

13.1 Helpers customizados

Quando criamos um controller, o Rails automaticamente cria um helper para esse controller em **app/helpers/**. Todo método escrito num helper, estará automaticamente disponível em sua view. Existe um Helper especial, o **application_helper.rb**, cujos métodos ficam disponíveis para todas as views.

13.2 Exercícios: Formatando valores monetários

1) Crie um método no nosso Application Helper para converter um número para valor monetário:

   a) Abra o arquivo **app/helpers/application_helper.rb**

   b) Adicione o seguinte método:

```
def valor_formatado(number)
  number_to_currency number,
    unit: "R$",
    separator: ",",
    delimiter: "."
end
```

   c) Em **app/views/qualificacoes/index.html.erb** e **app/views/qualificacoes/show.html.erb**, troque o seguinte código:

```
@qualificacao.valor_gasto
```

   por:

```
valor_formatado(@qualificacao.valor_gasto)
```

2) Teste nossa melhoria acessando **http://localhost:3000/qualificacoes/** e visualizando uma qualificação em específico.

13.3 Partials e a opção locals

A forma como o usuário cria comentários agora é claramente contra indicativa. O usuário tem que indicar o id do restaurante que irá comentar e que é um restaurante e não uma qualificação.

Para melhorar a usabilidade iremos adicionar uma caixa de texto na página de visualização do restaurante, dessa forma, o usuário poderá comentar o restaurante da própria página do restaurante.

Partial para formulário de comentário

Para criar o formulário teríamos que adicionar o seguinte código erb na view show do restaurante:

```
<%= form_for comentario.new do |f| %>
  <%= f.hidden_field :comentavel_id, value: @restaurante.id %>
  <%= f.hidden_field :comentavel_type, value: @restaurante.class %>

  <%= f.label :conteudo %><br />
  <%= f.text_area :conteudo %>

  <%= f.submit %>
<% end %>
```

Essa alteração serviria para o caso do usuário poder comentar só o restaurante, porém no nosso sistema, uma qualificação também pode ser comentada. Logo, precisaríamos do mesmo formulário na view show de qualificação.

Logo, a solução que nos permitiria reaproveitar o erb do formulário seria extrair o formulário para um partial:

```
<%= form_for comentario.new do |f| %>
  <%= f.hidden_field :comentavel_id, value: @restaurante.id %>
  <%= f.hidden_field :comentavel_type, value: @restaurante.class %>

  <%= f.label :conteudo %><br />
  <%= f.text_area :conteudo %>

  <%= f.submit %>
<% end %>
```

Porém, esse nosso partial faz uso da variável `@restaurante`, que não existe na view **show** de uma qualificação. Precisamos de um partial mais genérico, que se utilize de um comentável qualquer. Porém com o conhecimento que temos até agora isso não é possível.

Partial mais flexível com a opção locals

Para termos nosso partial genérico, iremos utilizar a opção `locals`. Com ela podemos passar algo parecido com um parâmetro. O `locals` é utilizado na chamada do método render.

Nas view **show** de **restaurantes** iremos renderizar o partial da seguinte maneira:

```
<%= render partial: "comentarios/novo_comentario",
           locals: {comentavel: @restaurante} %>
```

Já na view **show** de **qualificacoes** iremos passar a variável `@qualificacao`:

```
<%= render partial: "comentarios/novo_comentario",
           locals: {comentavel: @qualificacao} %>
```

Dessa forma, teremos disponível no nosso partial, uma variável `comentavel` e para utilizar essa variável iremos mudar o conteúdo do partial para:

```
<%= form_for Comentario.new do |f| %>
    <%= f.hidden_field :comentavel_id, value: comentavel.id %>
    <%= f.hidden_field :comentavel_type, value: comentavel.class %>

    <%= f.label :conteudo %><br />
    <%= f.text_area :conteudo %>

    <%= f.submit %>
<% end %>
```

Agora sim, nosso partial está flexível o suficiente para ser utilizado nas duas situações. Poderíamos melhorar ainda mais o código, encapsulando a chamada ao `render` em um helper:

```
def novo_comentario(comentavel)
  render partial: "comentarios/novo_comentario",
         locals: {comentavel: comentavel}
end
```

Dessa forma poderíamos renderizar o partial simplesmente chamando o helper:

```
<%= novo_comentario @restaurante %>
```

Ou no caso da qualificação:

```
<%= novo_comentario @qualificacao %>
```

13.4 Exercícios: Formulário para criar um novo comentário

Criar comentários usando o formulário padrão é algo pouco usual. Por isso tornaremos possível criar um comentário para um restaurante ou uma qualificação, apartir de sua página de visualização.

Ou seja, ao visualizar um restaurante ou qualificação, poderemos comentá-los através de um formulário.

1) Como utilizaremos o mesmo formulário nas views **show.html.erb** do restaurante e da qualificação, iremos escrever o ERB e uma partial.

   a) Crie um novo partial **app/views/comentarios/_novo_comentario.html.erb**.

   b) Insira o seguinte conteúdo:

```
<%= form_for Comentario.new do |f| %>
  <%= f.hidden_field :comentavel_id, value: comentavel.id %>
  <%= f.hidden_field :comentavel_type, value: comentavel.class %>

  <%= f.label :conteudo %><br />
  <%= f.text_area :conteudo %>

  <%= f.submit %>
<% end %>
```

2) a) Abra o arquivo **app/helpers/application_helper.rb**

   b) Vamos criar um novo método nesse *Helper*. A função desse método será renderizar um *partial* contendo o formulário para criação de novos comentários. Para isso, adicione o método `novo_comentario` na sua classe `ApplicationHelper`:

```
def novo_comentario(comentavel)
  render partial: "comentarios/novo_comentario",
         locals: {comentavel: comentavel}
end
```

3) Posicionaremos o formulário após as informações do restaurante.

   a) Abra o arquivo `app/views/restaurantes/show.html.erb` e após o último parágrafo adicione a chamada para o nosso novo *Helper*:

```
<%= novo_comentario @restaurante %>
```

4) Repita os passos da questão anterior com o arquivo `app/views/qualificacoes/show.html.erb`. Agora usando a variável `@qualificacao`.

5) Para testar iremos criar um comentário pelo formulário e verificar na listagem de comentários se ele foi realmente criado.

   a) Acesse **http://localhost:3000/restaurantes/**.

b) Entre na página de visualização de um restaurante.

c) Comente e visualize na listagem de comentários se ele foi criado corretamente.

6) Teste o formulário para uma qualificação também.

13.5 Partials de coleções

Nosso primeiro passo será possibilitar a inclusão da lista de comentários nas páginas de qualquer modelo que seja comentável. Para não repetir este código em todas as páginas que aceitem comentários, podemos isolá-lo em um partial:

```
<!-- app/views/comentarios/_comentario.html.erb -->
<p>
  <%= comentario.conteudo %> -
  <%= link_to '(remover)', comentario, :method => :delete %>
</p>
```

Perceba que não queremos renderizar esse partial uma única vez. Para cada comentário na array, devemos renderizar o partial, no final teremos um HTML com vários parágrafos.

Para fazer algo do tipo, teríamos que ter um ERB como:

```
<% @restaurante.comentarios.each do |comentario| %>
  <%= render partial: "comentarios/comentario",
        locals: {comentario: comentario} %>
<% end %>
```

Podemos simplificar o código acima, utilizando a opção `:collection`. Dessa maneira, o partial é renderizado uma vez para cada elemento que eu tenha no meu array:

```
<%= render partial: "comentarios/comentario",
      collection: @restaurante.comentarios %>
```

Perceba que nosso objetivo aqui é *renderizar* uma coleção. Estamos usando algumas convenções do *Rails* que irão permitir simplificar ainda mais o código acima. Além de criar uma variável com o o mesmo nome do partial que vamos *renderizar*, no nosso exemplo essa variável será chamada `comentario`, o *Rails* também será capaz de decidir **qual** é o arquivo partial que deve ser utilizado para exibir os itens de uma coleção.

Como temos uma coleção de **comentários** para exibir, basta invocar o método `render` passando a coleção como parâmetro, e o *Rails* cuidará do resto. Assim será possível omitir tanto o nome do *partial* como o *hash* que passamos como parâmetro anteriormente. Vamos fazer essa simplificação em nosso código e além disso garantir que a coleção seja *renderizada* apenas se não estiver vazia, dessa forma:

```
<%= render @restaurante.comentarios %>
```

13.6 Exercícios: Listando os comentários de um comentável

Assim como o formulário, a listagem tem que ser implementada de forma que seja reaproveitada nas views dos restaurantes e das qualificações, através de partials.

1) Crie o arquivo **app/views/comentarios/_comentario.html.erb** com o seguinte conteúdo:

```
<p>
  <%= comentario.conteudo %> -
  <%= link_to '(remover)', comentario, :method => :delete %>
</p>
```

2) a) Crie o arquivo **app/views/comentarios/_comentarios.html.erb**.

   b) Seu conteúdo deve ser:

```
<div id="comentarios">
  <h3>Comentários</h3>
  <% if comentarios.empty? %>
    Nenhum comentário
  <% else %>
    <%= render comentarios %>
  <% end %>
</div>
```

3) a) Abra o arquivo **app/helpers/application_helper.rb**

   b) Vamos adicionar um helper para agilizar a renderização do partial da listagem de comentários:

```
def comentarios(comentavel)
  render partial: "comentarios/comentarios",
    locals: {comentarios: comentavel.comentarios}
end
```

4) a) Abra a view de show dos restaurantes **app/views/restaurantes/show.html.erb**.

   b) Antes da chamada para `novo_comentario` adicione a seguinte linha:

```
<%= comentarios @restaurante %>
```

   c) Repita o exercício acima para a views de show das qualificacoes **app/views/qualificacoes/show.html.erb**. Lembrando de usar `@qualificacao` ao invés de `@restaurante`.

5) Após essas alterações deve ser possível ver os comentários na página de visualização de um restaurante ou de uma qualificação. Inicie o servidor e verifique.

13.7 Layouts

Como vimos, quando criamos um Partial, precisamos declará-lo em todas as páginas que desejamos utilizá-los. Existe uma alternativa melhor quando desejamos utilizar algum conteúdo estático que deve estar presente em todas as páginas: o **layout**.

Cada controller pode ter seu próprio layout, e uma alteração nesse arquivo se refletirá por todas as views desse controller. Os arquivos de layout devem ter o nome do controller, por exemplo **app/views/layouts/restaurantes.html.erb**.

Um arquivo de layout "padrão" tem o seguinte formato:

```
<html>
  <head>
    <title>Um título</title>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

Tudo o que está nesse arquivo pode ser modificado, com exceção do `<%= yield %>`, que renderiza cada view do nosso controlador.

Podemos utilizar ainda o layout **application.html.erb**. Para isso, precisamos criar o arquivo **app/views/layouts/application.html.erb** e apagar os arquivos de layout dos controladores que queremos que utilizem o layout do controlador application. Com isso, se desejarmos ter um layout único para toda nossa aplicação, por exemplo, basta ter um único arquivo de layout.

13.8 Exercícios: Criando um menu de navegação principal

1) Vamos agora criar um menu no nosso **Layout**, para poder navegar entre Restaurante, Cliente e Qualificação sem precisarmos digitar a url:

   a) Abra o layout mais genérico: **"app/views/layouts/application.html.erb"**

   b) Dentro do `<body>` e antes do `yield` digite o ERB da lista de opções do menu:

```
<body>
  <ul>
    <% %w[clientes qualificacoes restaurantes].each do |controller_name| %>
      <li>
        <%= link_to controller_name, controller: controller_name %>
      </li>
    <% end %>
```

```
    </ul>

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

2) Teste a navegação pelo menu: **http://localhost:3000/restaurantes**

13.9 Exercícios Opcionais: Menu como partial

1) Seria interessante podermos criar outros menus no decorrer do nosso sistema, para facilitar é necessário que extraiamos o ERB do menu para um **Partial**.

   a) Crie o partial "menu_principal": **"app/views/layouts/_menu_principal.html.erb"**

   b) Digite o seguinte ERB:

```
<ul>
  <% opcoes.each do |controller_name| %>
    <li>
      <%= link_to controller_name, controller: controller_name %>
    </li>
  <% end %>
</ul>
```

2) Perceba que a variável `opcoes` não existe, isso ocorre, por que vamos dizer qual será o valor dessa variável ao renderizar o partial, utilizando a opção `locals`:

   a) Abra o layout mais genérico: (**app/views/layouts/application.html.erb**)

   b) Substitua a lista de opções por uma chamada ao método `<%=render %>`, renderizando o partial **_menu_principal.html.erb**, dessa maneira:

```
...
<body>
  <%= render "menu_principal",
    locals: {opcoes: %w(restaurantes clientes qualificacoes)} %>

  <%= yield %>
</body>
...
```

3) Teste a navegação pelo menu: **http://localhost:3000/restaurantes**

4) Para simplificar a renderização do partial, podemos criar um helper que irá fazer esse trabalho:

   a) Abra o Helper "applications": **"app/helpers/application_helper.rb"**

   b) Digite o seguinte método:

```
def menu_principal(controllers)
  render partial: "menu_principal", locals: {opcoes: controllers}
end
```

5) a) Abra o layout mais genérico: (**app/views/layouts/application.html.erb**)

b) Substitua a chamada ao método `<%=render %>` por uma chamada ao helper **menu**:

```
...
<body>
  <%= menu_principal %w(restaurantes clientes qualificacoes) %>

  <%= yield %>
</body>
...
```

13.10 Aplicar CSS em apenas algumas views

Nossa listagem de comentários está com uma aparência muito desagradável, por isso iremos aplicar um CSS especial nas páginas que utilizarem nosso partial com os comentários.

É boa prática no desenvolvimento front-end, que importemos o CSS sempre dentro da tag `<head>`, porém se importarmos o CSS no **application.html.erb** esse CSS será importado desnecessariamente em todas as views de nossa aplicação.

Precisamos implementar nossas views de forma que a importação do nosso novo CSS seja feita somente se o partial **_comentarios.html.erb** for utilizado. Para fazer isso, teremos que utilizar o `yield` de uma forma diferente.

Primeiramente, iremos adicionar uma chamada ao `yield :css` dentro de **app/views/layouts/application.html.erb**, fazendo isso, estaremos tornando possível para a view inserir um conteúdo chamado `:css` naquela posição do layout. Nosso **application.html.erb** deve conter o seguinte ERB:

```
<head>
  <!-- (...) -->

    <%= stylesheet_link_tag "application", :media => "all" %>
    <%= yield :css %>

  <!-- (...) -->
</head>
```

Fazendo isso, tornamos possível que uma view ou partial possa inserir um conteúdo chamado `:css`. Agora precisamos fazer com que o partial **_comentarios.html.erb** defina um conteúdo `:css` importando o nosso CSS especial. Para isso, iremos chamar o método `content_for` em qualquer lugar do partial, dessa forma:

```
<% content_for :css do %>
    <link rel="stylesheet" href="/stylesheets/comentarios.css">
<% end %>
```

Após adicionar essas duas chamadas, nosso CSS será aplicado somente nas views que utilizam o partial **_comentarios.html.erb**.

13.11 Exercícios: Aplicando CSS para os comentários

Vamos configurar nosso layout e partials de forma que o CSS que irá estilizar os comentários só seja aplicado nas views necessárias:

1) Como nosso foco de estudo é o Rails, iremos utilizar um CSS que já está pronto. Entre na pasta **caelum** que está em seu Desktop e copie o arquivo: **caelum/71/comentarios.css** para dentro do nosso projeto **vota_prato/public/stylesheets/comentarios.css**.

2) Agora que temos o arquivo **public/stylesheets/comentarios.css** em nosso projeto vamos, usar o Rails para que esse CSS seja aplicado somente nas views onde o partial **_comentarios.html.erb** for renderizado:

   a) Abra o partial **app/views/comentarios/_comentarios.html.erb**.

   b) Ao final do arquivo, vamos invocar o helper `content_for` envolvendo a tag `link` associada ao nosso novo CSS:

   ```
   <% content_for :css do %>
     <link rel="stylesheet" href="/stylesheets/comentarios.css">
   <% end %>
   ```

3) Vamos agora indicar em que parte do layout devem ser inseridos os conteúdos referenciados pelo símbolo `:css`.

   a) Vamos abrir o layout mais genérico (**app/views/layouts/application.html.erb**).

   b) Adicione uma chamada ao `yield` dentro do `<head>` e logo abaixo da chamada ao helper `stylesheet_link_tag`:

   ```
   <%= stylesheet_link_tag "application", :media => "all" %>
   <%= yield :css %>
   ```

4) Teste o novo estilo da listagem de comentários acessando: **http://localhost:3000/restaurantes/** e visualizando a página de um restaurante.

Capítulo 14

Ajax com Rails

"O Cliente tem sempre razão"
– Selfridge, H.

Nesse capítulo, você verá como trabalhar com AJAX de maneira não obstrusiva no Rails.

14.1 Utilizando AJAX para remoção de comentários

Se observarmos a listagem de comentários perceberemos que a experiência do usuário não é agradável, após remover um comentário o usuário é redirecionado para a listagem de comentários. Podemos utilizar AJAX para uma experiência mais marcante no uso do site pelos usuários.

link_to realizando requisições AJAX

Ao remover um comentário, queremos que o usuário continue na página, porém iremos remover o parágrafo do comentário excluido. Primeiramente, teremos que fazer com que o link **(remover)** realize uma requisição AJAX ao invés de uma requisição convencional, impedindo assim que o usuário seja lançado para outra página.

Ao abrirmos o nosso partial **app/views/comentarios/_comentario.html.erb** veremos que há uma chamada ao helper `link_to`, que é responsável por criar o link **(remover)**. Esse helper, já nos disponibiliza uma opção para realização de uma requisição AJAX. Para obter esse comportamento, devemos utilizar a opção `remote: true`, da seguinte forma:

```
1 <!-- /app/views/comentarios/_comentario.html.erb -->
2 <p>
3   <%= comentario.conteudo %> -
```

```
  <%= link_to '(remover)', comentario, method: 'delete',
        remote: true %>
</p>
```

Após realizar essa alteração, poderemos remover um comentário sem sair da página atual. Porém o comentario em si continuará na tela. Se recarregarmos a página o comentário não aparecerá pois ele realmente foi excluido no banco de dados.

Utilizando JavaScript o parágrafo do comentário excluido

A action `ComentariosController#destroy` está sendo chamada de forma assíncrona (*Ajax*), porém a página não foi atualizada. Por isso precisaremos utilizar um código JavaScript que irá apagar o comentário logo após a action **destroy** responder a requisição AJAX com sucesso.

É uma boa prática no desenvolvimento front-end que importemos os nossos JavaScripts sempre antes de `</body>`. Para fazer isso, utilizaremos uma solução muito parecida com a do CSS, utilizando `yield` e `content_for`.

Vamos começar alterando o **app/views/layouts/application.html.erb** para que as nossas views e partials possam inserir um conteúdo JavaScript antes do **</body>**

```
<!-- /app/views/layouts/application.html.erb -->

    <!-- (...) -->

    <script type="text/javascript">
      <%= yield :js %>
    </script>
  </body>
</html>
```

Utilizaremos um código JavaScript que irá apagar o comentário assim que a requisição AJAX for respondida como bem sucedida:

```
$('#remove_comentario_<%= comentario.id %>').bind('ajax:success', function(){
  $('#comentario_<%= comentario.id %>').remove();
}
);
```

Agora vamos alterar o arquivo *app/views/comentarios/_comentario.html.erb* para colocar esse código JavaScript no lugar do 'yield :js'.

```
<p id="comentario_<%= comentario.id %>">
  <%= comentario.conteudo %>
```

```
  <%= link_to 'remover', comentario, :method => :delete,
          :remote => true,
          :id => "remove_comentario_#{comentario.id}"%>
</p>

<% content_for :js do%>
  $('#remove_comentario_<%= comentario.id %>').bind('ajax:success', function(){
    $('#comentario_<%= comentario.id %>').remove();
  });
<%end%>
```

Perceba que também adicionamos **id** no parágrafo e no link, adicionamos esse atributo pois o nosso código JavaScript necessita que cada comentário tenha um parágrafo com id **comentario_ID-DO-COMENTARIO** e um link com id **remove_comentario_ID-DO-COMENTARIO**.

Dessa forma, teremos o HTML gerado será algo como:

```
<div id='comentarios'><h3><Comentarios></h3>
  <p id="comentario_1">
    comentario 1
    <a href="/comentarios/1" data-method="delete" data-remote="true"
      id="remove_comentario_1" rel="nofollow">remover</a>
  </p>

  <p id="comentario_2">
    Comentario 2
    <a href="/comentarios/2" data-method="delete" data-remote="true"
      id="remove_comentario_2" rel="nofollow">remover</a>
  </p>

  <!-- Continuação do HTML -->

<script>
  $('#remove_comentario_1').bind('ajax:success', function(){
    $('#comentario_1').remove();
  });

  $('#remove_comentario_2').bind('ajax:success', function(){
    $('#comentario_2').remove();
  });
</script>
```

Após implementar, nossa página deve conter o código JavaScript ao final do código fonte da página. Para verificar, abra a página de visualização de um restaurante qualquer, pressione **CTRL + U** e verifique no código fonte da página se o nosso novo código JavaScript foi adicionado antes da tag body.

Mesmo com o código JavaScript sendo adicionado corretamente, o parágrafo do comentário ainda não será removido. Isso ocorrerá pois nosso código JavaScript assume que a resposta da requisição AJAX será de sucesso quando na verdade, ela é uma mensagem de falha pois não preparamos nosso controller para receber uma requisição AJAX.

Preparando o controller para uma requisição AJAX

Quando uma requisição AJAX é recebida pela action `ComentariosController#destroy`, a mesma é tratada como de formato **js** ao invés do convencional **html**.

Portanto, para preparar nossa action **destroy** basta adicionarmos um novo formato ao `respond_to` que só responda como bem sucedida. Para isso utilizaremos o método `head` passando o simbolo `:ok`:

```
1 def destroy
2   @comentario = Comentario.find(params[:id])
3   @comentario.destroy
4
5   respond_to do |format|
6     format.js { head :ok }
7   end
8 end
```

Após realizar todas essas alterações, poderemos finalmente desfrutar de um link remover que além de excluir no banco de dados, irá remover dinamicamente o parágrafo do comentário.

> **Para saber mais: AJAX em versões antigas do Rails**
>
> O Rails, antes da versão 3, tentava facilitar o desenvolvimento de código javascript com recursos como o RJS Templates, que produzia código javascript a partir de código Ruby.
>
> Contudo, o código gerado era acoplado ao Prototype, o que dificultava o uso de outras bibliotecas populares como o jQuery. No Rails 3 optou-se por essa nova forma, não obstrutiva, de se trabalhar com javascript, permitindo que os desenvolvedores tenham controle absoluto do código criado, e podendo, inclusive, escolher a biblioteca que será usada (Prototype, jQuery, Moo-tools ou qual quer outra, desde que um driver exista para essa biblioteca).
>
> Outra mudança interessante que mostra que o *Rails* é um framework em constante evolução foi a adoção do framework *jQuery* como padrão a partir da versão 3.1.

14.2 Exercícios: Link de remover comentário usando AJAX

1) Vamos configurar o link **remover** de forma que ele realize uma requisição AJAX.

a) Abra o partial que renderiza um único comentário (**app/views/comentarios/_comentario.html.erb**):

b) Adicione a opção **remote** com o valor **true** na chamada ao `link_to`:

```
<p>
  <%= comentario.conteudo %> -
  <%= link_to '(remover)', comentario, :method => :delete,
        :remote => true %>
</p>
```

2) Apesar de ser feita uma requisição AJAX, o comentário continua aparecendo. Vamos adicionar um código JavaScript que irá apagar o parágrafo quando o link for clicado:

a) Abra novamente o partial que renderiza um único comentário (**app/views/comentarios/_comentario.html.erb**):

b) Altere-o adicionando o código JavaScript e os **id**s que ele necessita. Ao final, o partial deve estar parecido com o código abaixo:

```
<p id="comentario_<%= comentario.id %>">
  <%= comentario.conteudo %> -
  <%= link_to '(remover)', comentario, :method => :delete,
        :remote => true,
        :id => "remove_comentario_#{comentario.id}" %>
</p>

<% content_for :js do %>
  $('#remove_comentario_<%= comentario.id %>').bind('ajax:success',
    function(){
      $('#comentario_<%=comentario.id%>').remove();
    }
  );
<% end %>
```

3) Como queremos que todo o JavaScript esteja posicionado antes do `</body>` é necessário que alteremos o layout `application.html.erb`.

a) Abra o arquivo **app/views/layouts/application.html.erb**.

b) Adicione as seguintes linhas logo abaixo do `<%= yield %>`:

```
<script>
  <%= yield :js %>
</script>
```

4) Precisamos também preparar a action **destroy** do nosso controller de comentários para responder à requisições AJAX.

a) Altere as seguintes linhas da action **destroy** do controller **comentarios_controller.rb**

```
def destroy
  @comentario = Comentario.find(params[:id])
  @comentario.destroy

  respond_to do |format|
    format.js { head :ok }
  end
end
```

5) Teste em **http://localhost:3000/restaurantes**, selecionando um restaurante e removendo um de seus comentários.

14.3 Adicionando comentários dinamicamente

Ao criar um novo comentário, fica claro que temos o mesmo problema que tinhamos no link **(remover)**. Após criar um comentário somos redirecionados para uma outra página. Portanto resolveremos esse problema fazendo com que o formulário envie uma requisição AJAX.

Formulário enviando requisições AJAX

Assim como o `link_to`, o helper `form_for` também tem a opção `remote` para ser utilizadas em formulários que devem realizar requisições AJAX.

Logo, precisaremos realizar uma pequena alteração no nosso partial **app/views/comentarios/_novo_comentario.html.erb** para que o formulário do mesmo passe a realize requisições AJAX:

```
<!-- /app/views/comentarios/_novo_comentario.html.erb -->
<%= form_for Comentario.new, remote: true do |f| %>

<!-- (...) -->
```

Após utilizar a opção `remote: true`, conseguiremos criar um comentário, porém o mesmo não aparecerá na listagem de comentários.

JavaScript para atualizar a lista

Para atualizar a lista após a resposta da requisição AJAX iremos utilizar o seguinte código JavaScript:

```
$('form').bind('ajax:complete', function(){
  $('#comentarios').replaceWith(result.responseText);
  $('textarea').val("");
});
```

Vamos nos aproveitar do `yield :js` que já foi inserido no layout, ou seja, só precisaremos adicionar a chamada ao método `content_for` em qualquer local do nosso partial **app/views/comentarios/_novo_comentario.html.erb**:

```
<% content_for :js do %>
  $('form').bind('ajax:complete', function(){
    $('#comentarios').replaceWith(result.responseText);
    $('textarea').val("");
  });
<% end %>
```

Após realizar essas alterações verificaremos que ao tentar criar um comentário, a listagem de comentários não é atualizada. Ao invés disso, a listagem é substituida por uma outra página. Isso ocorrerá, pois nosso código JavaScript substitui a listagem pela resposta da requisição.

Isso ocorre pois nossa action `ComentariosController#create` não está preparada para responder com a listagem de comentários atualizada.

Preparando view para requisição AJAX

Até agora trabalhamos com arquivos `html.erb`, que são utilizados na resposta para requisições do formato **html**.

Porém, nossa action `ComentariosController#create` precisa responder com um conteúdo para o formato **js** que é o utilizado para requisições AJAX. Para fazer isso, iremos criar um arquivo de extensão **js.erb**. Como a action que irá responder é a **create** iremos criar o arquivo **app/views/comentarios/create.js.erb** que irá simplesmente renderizar a listagem de comentários novamente:

```
<%= comentarios @comentario.comentavel %>
```

Note que nossas views anteriores utilizavam o layout **application.html.erb**, porém nossa nova view irá utilizar o layout de acordo com seu próprio formato, ou seja, o arquivo **app/views/layouts/application.js.erb**. Como a resposta terá somente a lista e nada mais, nosso novo arquivo de layout deverá ter somente duas chamadas para `yield`. Uma para renderizar o conteúdo da view e a outra para renderizar o código JavaScript:

```
<%= yield %>

<script>
  <%= yield :js %>
</script>
```

Respondendo AJAX com as views preparadas

Após criar as novas views, precisaremos declarar na action **create** que ela deve dar suporte ao formato **js**. Para isso, basta adicionar uma chamada à `format.js` dentro do `respond_to`:

```
def create
  @comentario = Comentario.new params[:comentario]

  respond_to do |format|
    if @comentario.save
      format.js
      end
    end
end
```

Após realizar as alterações acima, poderemos criar comentários sem ter que esperar a página inteira ser carregada. Pois a listagem de comentários será atualizada dinamicamente.

14.4 Exercícios: AJAX no formulário de comentários

1) Primeiramente iremos configurar nosso formulário para criação de comentários de forma que ele realize requisições AJAX:

   a) Abra o partial do formulário para criação de comentários: (**app/views/comentarios/_novo_comentario.htm**

   b) Na chamada do `form_for` utilize a opção `remote` como `true`:

   ```
   <%= form_for Comentario.new, remote: true do |f| %>
   ```

   c) No final do partial, adicione o código JavaScript que irá lidar com a resposta da requisição AJAX:

   ```
   <% content_for :js do %>
     $('form').bind('ajax:complete', function(xhr, result){
       $('#comentarios').replaceWith(result.responseText);
       $('textarea').val("");
     });
   <% end %>
   ```

2) Para finalizar o processo temos que alterar o nosso controller de comentários para responder com a lista de comentários do nosso comentável.

   a) Altere o método create no **app/controllers/comentarios_controller.rb**

   ```
   if @comentario.save
     format.js
   else
   ```

b) E agora crie o arquivo **views/comentarios/create.js.erb**

```
<%= comentarios @comentario.comentavel %>
```

c) Para que nosso partial exiba corretamente os javascripts precisamos criar um layout para colocar o bloco de yield dos javascripts.

d) Crie o arquivo **app/views/layouts/application.js.erb** com o seguinte conteúdo

```
<%= yield %>

<script>
  <%= yield :js %>
</script>
```

e) Agora já é possível cadastrar um comentário direto do show dos comentáveis

Capítulo 15

Algumas Gems Importantes

"A minoria pode ter razão, a maioria está sempre errada"
– Mikhail Aleksandrovitch Bakunin

15.1 Engines

A partir do *Rails 3.1* foi criado um conceito para facilitar a extensão de uma aplicação chamado *Rails Engine*. Uma *engine* é uma espécie de mini aplicação *Rails* que permite isolar funcionalidades que podem ser reaproveitadas em qualquer aplicação *Rails* "comum" de uma forma estruturada e **plugável**.

Um bom exemplo de *Rails engine* é a *gem kaminari* que facilita muito a criação de uma lista paginada para exibição de registros. Basta adicionar a dependência do `Gemfile`, rodar o comando `bundle install` e pronto! Toda a funcionalidade de paginação será fornecida sem a necessidade de nenhum tipo de configuração.

Primeiro vamos editar o arquivo `Gemfile` para adicionar a dependência:

```
gem "kaminari"
```

Agora para que os dados sejam paginados a partir de uma busca no banco de dados, basta adicionar a invocação do método `page` do *kaminari*. Podemos usar essa funcionalidade na listagem de restaurantes como no código a seguir:

```
@restaurantes = Restaurante.all.page params['page']
```

O método `page` funciona como um finder normal. Suporta todas as opções previamente vistas, como `:conditions`, `:order` e `:include`. Note também que passamos um parâmetro para o método `page` que é a **página** a partir da qual queremos listar os registros. Esse parâmetro será gerado pelo próprio `kaminari` através de um *helper method* que invocaremos na view, mais adiante você verá como isso funciona.

O número de items por página é padronizado em 25, mas você pode customizar de duas formas. Através do método per_page nas classes `ActiveRecord::Base`:

```
class Restaurante < ActiveRecord::Base
  paginates_per 10
  # ...
end
```

Ou invocando o método `per` e passando como parâmetro o número máximo de registros por página. O método `per` está disponível no objeto criado pelo `kaminari` portanto só pode ser invocado após uma chamada ao método `page`. O código seguinte exemplifica o seu uso:

```
@restaurantes = Restaurante.all.page(params['page']).per(10)
```

E como dito, veja como usar o *helper method* do *kaminari* para gerar os links de paginação já com o parâmetro `page` que usamos durante os exemplos de paginação:

```
1 # index.html.erb
2 <% @restaurantes.each do |restaurante| %>
3   <li><%= restaurante.nome %></li>
4 <% end %>
5 <%= paginate @restaurantes %>
```

Uma coisa bem interessante sobre o `kaminari` é sua facilidade de customização. Você pode alterar o texto padrão usado por ele para geração de links, assim como a aparência desses mesmos links. Veja o repositório oficial para maiores detalhes: https://github.com/amatsuda/kaminari.

15.2 Exercícios: Paginação com Kaminari

1) Adicione o *kaminari* ao seu projeto, no arquivo `Gemfile`

   ```
   gem 'kaminari'
   ```

   Execute o seguinte comando no terminal para instalar a gem:

   ```
   $ bundle install
   ```

2) Abra o arquivo **app/controllers/restaurantes_controller.rb**. Na action index, troque a linha:

   ```
   @restaurantes = Restaurante.order("nome")
   ```

   por

   ```
   @restaurantes = Restaurante.order("nome").page(params['page']).per(3)
   ```

3) Abra o arquivo **app/views/restaurantes/index.html.erb**. Adicione a linha abaixo após o fechamento da *tag table* (`</table>`):

```
<%= paginate @restaurantes %>
```

4) Abra a listagem de restaurantes e verifique a paginação.

15.3 File Uploads: Paperclip

Podemos fazer upload de arquivos sem a necessidade de plugins adicionais, utilizando algo como o código abaixo:

```
File.open("public/"+path, "nome") do |f|
  f.write(params[:upload]['picture_path'].read)
end
```

O código acima recebe o binário do arquivo e faz o upload para a pasta public. Porém, ao fazer um upload seria interessante fazer coisas como redimensionar a imagem, gerar thumbs, associar com models ActiveRecord, etc.

Um dos primeiros plugins rails voltados para isso foi o *attachment_fu*. Hoje em dia o plugin mais indicado é o **Paperclip**. O Paperclip tem como finalidade ser um plugin de fácil uso com o modelos Active Record. As configurações são simples e é possível validar tamanho do arquivo ou tornar sua presença obrigatória. O paperclip tem como pré-requisito o *ImageMagick*,

Economizando espaço em disco

É boa prática ao configurar os tamanhos das imagens que armazenaremos, sobrescrever o tamanho do estilo "original", dessa maneira evitamos que o Paperclip salve imagens muito maiores do que utilizaremos em nossa aplicação.

```
has_attached_file :foto, styles: {
    medium: "300x300>",
    thumb: "100x100>",
    original: "800x600>"
}
```

No exemplo acima, a imagem "original" será salva com tamanho máximo de 800x600 pixels, isso evita que armazenemos desnecessariamente em nossa aplicação, por exemplo, imagens de 4000x3000 pixels, tamanho resultante de uma foto tirada em uma máquina fotográfica digital de 12MP.

15.4 Exercícios: Possibilitando upload de imagens

1) Para instalar o paperclip, abra o arquivo Gemfile e adicione a gem:

```
gem 'paperclip'
```

E no terminal, rode :

```
bundle install
```

2) Habilite um restaurante à receber um parâmetro **foto** e adicione o **has_attached_file** do paperclip na classe Restaurante. Vamos configurar mais uma opção que daremos o nome de *styles*. Toda vez que a view chamar a foto do restaurante com essa opção, o Rails buscará pelo thumb.

```
class Restaurante < ActiveRecord::Base
  attr_accessible :endereco, :especialidade, :nome, :foto

  # .. validações e relacionamentos

  has_attached_file :foto, styles:
    { medium: "300x300>", thumb: "100x100>" }
end
```

3) a) Precisamos de uma migration que defina novas colunas para a foto do restaurante na tabela de restaurantes. O paperclip define 4 colunas básicas para nome, conteúdo, tamanho do arquivo e data de update.O Paperclip nos disponibiliza um gerador para criar esse tipo de migration, para utilizá-lo execute no terminal:

```
rails generate paperclip Restaurante foto
```

b) Rode a migration no terminal com:

```
rake db:migrate
```

4) Abra a view **app/views/restaurantes/_form.html.erb** e altere o formulário. Seu form deve ficar como o abaixo:

```
<%= form_for @restaurantes, html: {multipart: true} do |f| %>
  <!--outros campos-->
  <%= f.file_field :foto %>
<% end %>
```

5) Abra a view **app/views/restaurantes/show.html.erb** e adicione:

```
<p>
  <b>Foto:</b>
    <%= image_tag @restaurante.foto.url(:thumb) %>
</p>
```

Repare que aqui chamamos o thumb, que foi configurado como um dos styles do model. Suba o server e insira um novo restaurante com foto.

15.5 Nokogiri

Nokogiri é uma biblioteca poderosa para manipulação de xhtml. Bastante útil para capturar conteúdo da internet que não tenha sido criado pensando em integração e não oferece formatos mais adequados para serem consumidos por outros sistemas, como json ou xml.

```
gem install nokogiri
```

`open-uri` é uma biblioteca que usaremos para fazer requisições http:

```
doc = Nokogiri::HTML(open('https://twitter.com/caelum'))
```

Analisando o html gerado pelo twitter, vemos que os tweets estão sempre dentro de elementos com a classe `"tweet"`. Além disso, dentro de cada `tweet`, a única parte que nos interessa é o conteúdo dos subitems de classe `"js-tweet-text"`, onde encontraremos as mensagens.

Podemos procurar estes itens com o Nokogiri, usando seletores *CSS*. Expressões *XPath* também poderiam ser usadas:

```
doc.css ".tweet .js-tweet-text"
```

Para imprimir cada um dos itens de uma maneira mais interessante:

```
items = doc.css ".tweet .js-tweet-text"
items.each do |item|
  puts item.content
end
```

15.6 Exercícios: Extraindo dados do Twitter

1) Vamos fazer um leitor de tweets de um determinado usuário

   a) Crie um arquivo chamado **"twitter_reader.rb"**

   b) Adicione ás seguintes linhas:

```
require 'rubygems'
require 'open-uri'
require 'nokogiri'

doc = Nokogiri::HTML(open('https://twitter.com/caelum'))
items = doc.css ".content"
items.each do |item|
```

```
            autor = item.css(".fullname").first.content
    tweet = item.css(".js-tweet-text").first.content

            puts autor
            puts tweet
            puts
end
```

c) Teste usando o comando `ruby twitter_reader.rb`

Capítulo 16

Apêndice: Testes

"Ninguém testa a profundidade de um rio com os dois pés."
– Provérbio Africano

16.1 O Porquê dos testes?

Testes de Unidade são classes que o programador desenvolve para se certificar que partes do seu sistema estão funcionando corretamente.

Eles podem testar validações, processamento, domínios etc, mas lembre-se que um teste unitário deve testar *somente um pedaço de código* (de onde veio o nome *unitário*).

Criar esse tipo de testes é uma das partes mais importantes do desenvolvimento de uma aplicação pois possibilita a verificação real de todas as partes do programa automaticamente.

> **Extreme Programming (XP)**
>
> Extreme Programming é um conjunto de práticas de programação que visam a simplicidade, praticidade, qualidade e flexibilidade de seu sistema. Os testes de unidade fazem parte dessa metodologia de programação.

O Ruby já possui classes que nos auxiliam no desenvolvimento destes testes.

16.2 `Test::Unit`

`Test::Unit` é a biblioteca usada para escrever suas classes de teste.

Ao escrever testes em Ruby utilizando esse framework, você deve herdar a classe `TestCase` que provê a funcionalidade necessária para fazer os testes.

```
require 'test/unit'

  class PessoaTest < Test::Unit::TestCase
    # ...
end
```

Ao herdar `Test`*`Unit`*`TestCase`, você ganha alguns métodos que irão auxiliar os seus testes:

- `assert(boolean, msg=nil)`
- `assert_equal(esperado, atual, msg=nil)`
- `assert_not_equal(esperado, atual, msg=nil)`
- `assert_in_delta(esperado, atual, delta, msg=nil)`
- `assert_instance_of(classe, objeto, msg=nil)`
- `assert_kind_of(classe, objeto, msg=nil)`
- `assert_match(regex, texto, msg=nil)`
- `assert_no_match(regex, texto, msg=nil)`
- `assert_nil(objeto, msg=nil)`
- `assert_not_nil(objeto, msg=nil)`
- `assert_respond_to(objeto, metodo, msg=nil)`
- `assert_same(esperado, atual, msg=nil)`
- `assert_not_same(esperado, atual, msg=nil)`

O método `assert` simples recebe como parâmetro qualquer expressão que devolva um valor booleano e todos os métodos `assert` recebem opcionalmente como último argumento uma mensagem que será exibida caso a asserção falhe.

Mais detalhes e outros métodos `assert` podem ser encontrados na documentação do módulo `Test`*`TestUnit`*`Assertions`, na documentação da biblioteca core da linguagem Ruby (http://ruby-doc.org/core/).

Os testes podem ser executados em linha de comando, bastando chamar `ruby o_que_eu_quero_testar.rb`. O resultado é um **".**

para os testes que passarem, **"E"** para erros em tempo de execução e **"F"** para testes que falharem.

Também é possível executar todos os testes com algumas tasks do rake:

```
# roda todos os testes de unidade, de integração e funcionais
rake test

# roda todos os testes da pasta test/unit
rake test:units

# roda todos os testes da pasta test/functional
rake test:functionals

# roda todos os testes da pasta test/integration
rake test:integration

# roda todos os testes de plugins, na pasta vendor/plugins
rake test:plugins
```

Existem ainda outras tarefas disponíveis para o rake. Sempre podemos consultá-las com `rake -T`, no diretório do projeto.

Podemos criar uma classe de teste que só possua um único "assert true", no diretório test/unit/.

```
class MeuTeste < Test::Unit::TestCase
    def test_truth
        assert true
    end
end
```

Ao escrever testes de unidade em projetos Ruby On Rails, ao invés de herdar diretamente de `Test::Unit::TestCase`, temos a opção de herdar da classe fornecida pelo ActiveSupport do Rails:

```
require 'test_helper'

class RestauranteTest < ActiveSupport::TestCase
  def test_anything
    assert true
  end
end
```

Além disso, todos os testes em projetos Rails devem carregar o arquivo **test_helper.rb**, disponível em qualquer projeto gerado pelo Rails. As coisas comuns a todos os testes, como método utilitários e configurações, ficam neste arquivo.

A vantagem de herdar de `ActiveSupport::TestCase` ao invés da original é que o Rails provê diversas funcionalidades extras aos testes, como fixtures e métodos `assert` extras. Alguns dos asserts extras:

- `assert_difference`

- `assert_no_difference`
- `assert_valid(record)` - disponível em testes de unidade
- `assert_redirected_to(path)` - para testes de controladores
- `assert_template(esperado)` - também para controladores
- entre outros

16.3 Exercícios - Teste do modelo

1) Vamos testar nosso modelo restaurante. Para isso precisamos utilizar o banco de dados específico para testes.

```
rake db:create:all
rake db:migrate RAILS_ENV=test
```

2) O Rails já tem suporte inicial para testes automatizados. Nossa aplicação já possui arquivos importantes para nossos testes. Abra o arquivo **test/fixtures/restaurantes.yml**. Esse arquivo simula os dados de um restaurante. Crie os dois restaurantes abaixo. **Importante:** Cuidado com a identação!

```
fasano:
  nome: Fasano
  endereco: Rua Vergueiro

fogo_de_chao:
  nome: Fogo de Chao
  endereco: Avenida dos Bandeirantes
```

3) O arquivo de teste do modelo está em **test/unit/restaurante_test.rb**.

```
require 'test_helper'

class RestauranteTest < ActiveSupport::TestCase
  fixtures :restaurantes

    def test_restaurante

      restaurante = Restaurante.new(
                :nome => restaurantes(:fasano).nome,
                :endereco => restaurantes(:fasano).endereco,
                :especialidade => restaurantes(:fasano).especialidade)

      msg = "restaurante não foi salvo. "
            + "errors: ${restaurante.errors.inspect}"
      assert restaurante.save, msg
```

```
    restaurante_fasano_copia = Restaurante.find(restaurante.id)

    assert_equal restaurante.nome, restaurante_fasano_copia.nome
  end
end
```

4) Para rodar o teste, vá na raiz do projeto pelo terminal e digite:

```
rake test
```

Verifique se tudo está certo:

```
Loaded suite test/unit/restaurante_test
Started
.
Finished in 0.044991 seconds.

1 tests, 4 assertions, 0 failures, 0 errors
```

16.4 Exercícios - Teste do controller

1) Para testar o controller de restaurantes vamos criar uma nova action chamada **busca**. Essa action direciona para o restaurante buscado caso encontre ou devolve uma mensagem de erro caso contrário. Abra o arquivo **app/controllers/restaurantes_controller.rb** e adicione a action **busca**:

```
def busca
  @restaurante = Restaurante.find_by_nome(params[:nome])
  if @restaurante
    redirect_to :action => 'show', :id => @restaurante.id
  else
    flash[:notice] = 'Restaurante não encontrado.'
    redirect_to :action => 'index'
  end
end
```

2) Abra o arquivo **test/functional/restaurantes_controller_test.rb**.

```
require 'test/test_helper'

class RestaurantesControllerTest < ActionController::TestCase
  fixtures :restaurantes

    def test_procura_restaurante
      get :busca, :nome => 'Fasano'
      assert_not_nil assigns(:restaurante)
```

```
      assert_equal restaurantes(:fasano).nome, assigns(:restaurante).nome
      assert_redirected_to :action => 'show'
    end

  def test_procura_restaurante_nao_encontra
    get :busca, :nome => 'Botequin'
    assert_redirected_to :action => 'index'
    assert_equal 'Restaurante não encontrado.', flash[:notice]
  end
end
```

Verifique se tudo está certo;

```
Loaded suite test/functional/restaurantes_controller_test
Started
..
Finished in 0.206066 seconds.

2 tests, 4 assertions, 0 failures, 0 errors
```

3) Rode o teste no terminal com **rake test**.

16.5 RSpec

Muito mais do que uma nova forma de criar testes de unidade, RSpec fornece uma forma de criar especificações executáveis do seu código.

No TDD, descrevemos a funcionalidade esperada para nosso código através de testes de unidade. BDD (*Behavior Driven Development*) leva isso ao extremo e diz que nossos testes de unidade devem se tornar especificações executáveis do código. Ao escrever as especificações estaremos pensando no **comportamento esperado** para nosso código.

> ### Introdução ao BDD
>
> Uma ótima descrição sobre o termo pode ser encontrada no site do seu próprio criador: Dan North.
>
> http://dannorth.net/introducing-bdd/

RSpec fornece uma DSL (*Domain Specific Language*) para criação de especificações executáveis de código. As especificações do RSpec funcionam como exemplos de uso do código, que validam se o código está mesmo fazendo o que deveria e funcionam como documentação.

http://rspec.info

Para instalar o rspec e usar em qualquer programa Ruby, basta instalar o gem:

```
gem install rspec
```

Para usar em aplicações Rails, precisamos instalar mais um gem que dá suporte ao rspec ao Rails. Além disso, precisamos usar o gerador que vem junto desta gem, para adicionar os arquivos necessários nos projetos que forem usar rspec:

```
cd projeto-rails
rails generate rspec:install
```

O último comando também adiciona algumas tasks do rake para executar as specs do projeto, além de criar a estrutura de pastas e adicionar os arquivos necessários.

```
rake spec     # executa todas as specs do projeto
rake -T spec  # para ver as tasks relacionadas ao rspec
```

O rspec-rails também pode ser instalado como plugin, porém hoje é altamente recomendado seu uso como gem. Mais detalhes podem ser encontrados na documentação oficial

http://wiki.github.com/rspec/rspec-rails/

RSpec é compatível com testes feitos para rodar com `Test::Unit`. Desta forma, é possível migrar de forma gradativa. Apesar disso, a sintaxe oferecida pelo RSpec se mostra bem mais interessante, já que segue as ideias do Behavior Driven Development e faz com que os testes se tornem especificações executáveis do código:

```
describe Restaurante, " com nome" do
  it "should have name"
    Restaurante.all.should_not be_empty
    Restaurante.first.should_not be_nil
    Restaurante.first.name.should == "Fasano"
  end
end
```

A classe de teste vira um **Example Group** (`describe`). Cada método de teste vira um **Example** (`it "should ..."`).

Além disso, os métodos `assert` tradicionais do `Test::Unit` viram uma chamada de `should`. O RSpec adiciona a **todos** os objetos os métodos `should` e `should_not`, que servem para validarmos alguma condição sobre o estado dos nossos objetos de uma forma mais legível e expressiva que com asserts.

Como argumento para o método `should`, devemos passar uma instância de `Matcher` que verifica uma condição particular. O RSpec é extremamente poderoso, pois nos permite escrever nossos próprios Matchers. Apesar disso, já vem com muitos prontos, que costumam ser mais do que suficientes:

- be_<nome> para métodos na forma <nome>?.

```
# testa: objeto.empty?
objeto.should be_empty

# testa: not objeto.nil?
objeto.should_not be_nil

# testa: objeto.kind_of(Restaurante)
objeto.should be_kind_of(Restaurante)
```

  Além de be_<nome>, também podemos usar be_a_<nome> ou be_an_<nome>, aumentando a legibilidade.

- be_true, be_false, eql, equal, exist, include:

```
objeto.should be_true
objeto.should_not be_false

# testa: objeto.eql?(outro)
objeto.should eql(outro)

# testa: objeto.equal?(outro)
objeto.should equal(outro)

objeto.should exist  # testa: objeto.exist?
[4,5,3].should include(3)  # testa: [4,5,3].include?(3)
```

- have_<nome> para métodos na forma has_<nome>?.

```
itens = { :um => 1, :dois => '2' }

# testa: itens.has_key?(:dois)
itens.should have_key(:dois)

# testa: not itens.has_value?(/3/)
itens.should_not have_value(/3/)
```

- be_close, inclui tolerância.

```
conta = 10.0 / 3.0
conta.should be_close(3.3, 0.1) # == 3.3 ~0.1
```

- have(num).<colecao>, para testar a quantidade de itens em uma associação.

```
# testa categoria.produtos.size == 15
categoria.should have(15).produtos
```

  Um uso especial deste *matcher* é para objetos que já são coleções. Neste caso, podemos usar o nome que quisermos:

```
array = [1,2,3]

# testa array.size == 3
array.should have(3).items

# mesma coisa
array.should have(3).numbers
```

- `have_at_least(num).<colecao>`: mesma coisa que o anterior, porém usa >=.
- `have_at_most(num).<colecao>`: mesma coisa que o anterior, porém usa <=.
- `match`, para expressões regulares.

```
# verifica se começa com F
texto.should match(/^F/)
```

Este são os principais, mas ainda existem outros. Você pode encontrar a lista de Matchers completa na documentação do módulo `Spec::Matchers`:

Exemplos pendentes

Um exemplo pode estar vazio. Desta forma, o RSpec o indicará como pendente:

```
describe Restaurante do
  it "should have endereco"
end
```

Isto facilita muito o ciclo do BDD, onde escrevemos o teste primeiro, antes do código de verdade. Podemos ir pensando nas funcionalidades que o sistema deve ter e deixá-las pendentes, antes mesmo de escrever o código. Em outras palavras, começamos especificando o que será escrito.

Before e After

Podemos definir algum comportamento comum para ser executado antes ou depois de cada um dos exemplos, como o `setup` e o `teardown` do `Test::Unit`:

```
describe Restaurante do
  before do
    @a_ser_testado = Restaurante.new
  end

  it "should ..."

  after do
```

```
    fecha_e_apaga_tudo
  end
end
```

Estes métodos podem ainda receber um argumento dizendo se devem ser executados novamente para cada exemplo (`:each`) ou uma vez só para o grupo todo (`:all`):

```
describe Restaurante do
  before(:all) do
    @a_ser_testado = Restaurante.new
  end

  it "should ..."

  after(:each) do
    fecha_e_apaga_tudo
  end
end
```

16.6 Cucumber, o novo Story Runner

RSpec funciona muito bem para especificações em níveis próximos ao código, como as especificações unitárias.

User Stories é uma ferramenta indicada para especificações em níveis mais altos, como funcionalidades de negócio, ou requisitos. Seu uso está sendo bastante difundido pela comunidade Rails. User Stories costumam ter o seguinte formato:

```
In order to <benefício>
As a <interessado>
I want to <funcionalidade>.
```

Cucumber é uma excelente biblioteca escrita em Ruby, que serve para tornar especificações como esta, na forma de *User Stories*, escritas em texto puro, executáveis. Cucumber permite a associação de código Ruby arbitrário, usualmente código de teste com RSpec, a cada um dos passos desta descrição da funcionalidade.

Para instalar tudo o que é necessário:

```
gem install cucumber capybara database_cleaner
```

```
gem 'database_cleaner'
gem 'cucumber-rails'
gem 'cucumber'
```

```
gem 'rspec-rails'
gem 'spork'
gem 'launchy'
```

Para projetos rails, é possível usar o *generator* fornecido pelo Cucumber para adicionar os arquivos necessários ao projeto:

```
cd projetorails
rails generate cucumber:install
```

As User Stories são chamadas de **Features** pelo Cucumber. São arquivos de texto puro com a extensão *.feature*. Arquivos com definição de features sempre contém uma descrição da funcionalidade (**Story**) e alguns exemplos (**Scenarios**), na seguinte estrutura:

```
Feature: <nome da story>
  In order to <beneficio>
  As a <interessado>
  I want to <funcionalidade>

  Scenario: <nome do exemplo>
    Given <pré condições>
    And   <mais pré condições>
    When  <ação>
    And   <mais ação>
    Then  <resultado>
    And   <mais resultado>

  Scenario: <outro exemplo>
    Given ...
    When ...
    Then ...
```

Antigamente, o RSpec incluia sua própria implementação de Story Runner, que hoje está sendo substituída pelo Cucumber. O RSpec Story Runner original utilizava um outro formato para features, mais tradicional, que não dá prioridade ao *Return Of Investment*. O benefício da funcionalidade fica em segundo plano, no final da descrição:

```
Story: transfer from savings to checking account
  As a savings account holder
  I want to transfer money from my savings account to my checking account
  So that I can get cash easily from an ATM

  Scenario: ...
```

O importante para o Cucumber são os exemplos (**Scenarios**) que explicam a funcionalidade. Cada um dos Scenarios contém um conjunto de passos, que podem ser do tipo **Given** (pré-requisitos), **When** (ações), ou **Then** (resultado).

A implementação de cada um dos passos (*steps*) dos *scenarios* devem ficar dentro do diretório **step_definitions/**, na mesma pasta onde se encontram os arquivos *.feature*, texto puro.

O nome destes arquivos que contém a definição de cada um dos passos deve terminar com `_steps.rb`. Cada passo é representado na chamada dos métodos `Given`, `Then` ou `When`, que recebem como argumento uma **String** ou **expressão regular** batendo com o que estiver escrito no arquivo de texto puro (*.feature*).

Tipicamente, os projetos contém um diretório **features/**, com a seguinte estrutura:

```
projeto/
|-- features/
|   |-- minha.feature
|   |-- step_definitions/
|   |   |-- alguns_steps.rb
|   |   `-- outros_steps.rb
|   `-- support/
|       `-- env.rb
```

O arquivo **support/env.rb** é especial do Cucumber e sempre é carregado antes da execução dos testes. Geralmente contém a configuração necessária para os testes serem executados e código de suporte aos testes, como preparação do Selenium ou Webrat.

Os arquivos com definições dos passos são arquivos Ruby:

```
Given "alguma condição descrita no arquivo texto puro" do
  # código a ser executado para este passo
end

Given /e outra condicao com valor: (.*)/ do |valor|
  # código de teste para esse passo
end

When /alguma acao/
  # ...
end

Then /verifica resultado/
  # ...
end
```

O código de teste para cada passo pode ser qualquer código Ruby. É comum o uso do RSpec para verificar condições (métodos `should`) e **Webrat** ou **Selenium** para controlar testes de aceitação. Mais detalhes sobre estes frameworks para testes de aceitação podem ser vistos no capítulo *"Outros testes e specs"*.

Não é necessário haver um arquivo com definição de passos para cada arquivo de feature texto puro. Isto é até considerado má prática por muitos, já que inibe o reuso para definições de *steps*.

Capítulo 17

Apêndice: Rotas e Rack

"Não é possível estar dentro da civilização e fora da arte"
– Rui Barbosa

O modo como urls são ligadas a controladores e actions pode ser customizado no Rails. O módulo responsável por esta parte é o que foi criado com o seu projeto `NomeDoProjeto::Application.routes` e as rotas podem ser customizadas no arquivo config/routes.rb.

17.1 Rack

O `rack` é uma abstração das requisições e respostas HTTP da maneira mais simples possível. Criando uma API unificada para servidores, frameworks, e softwares (os conhecidos middleware) em apenas uma chamada de método.

A grande motivação da criação do Rack é que, diferente do mundo java onde existe uma especificação que abstrai todo o HTTP, no mundo ruby cada framework havia criado a sua forma de tratar as requisições e respostas. Por isso, escrever um servidor ou mesmo permitir que o framework X pudesse rodar em um servidor que já existisse era um trabalho realmente complicado. Graças ao surgimento do rack e da sua padronização hoje é possível que qualquer servidor que conheça rack consiga executar qualquer aplicação que se comunique com o HTTP através do rack.

Mais do que isso, hoje também é possível fazer uma "aplicação" web em apenas uma linha. Exemplo:

```
run Proc.new {|env| [200, {"Content-Type" => "text/html"},
                  ["Hello World"]]}
```

Basta salvar esse arquivo, por exemplo como **hello.ru**, e subir nosso servidor pelo Terminal com o seguinte comando:

```
$ rackup hello.ru
```

Para criar uma "aplicação" em rack tudo o que precisamos é criar um método que retorne [httpStatusCode, headers, body], como no exemplo acima.

O comando rackup é criado quando instalamos a gem 'rack' e serve para iniciar aplicações feitas em rack. Elas nada mais são que um arquivo ruby, mas devem ser salvos com a extensão .ru (RackUp) e devem chamar o método `run`.

17.2 Exercícios - Testando o Rack

1) Vamos fazer uma aplicação rack.

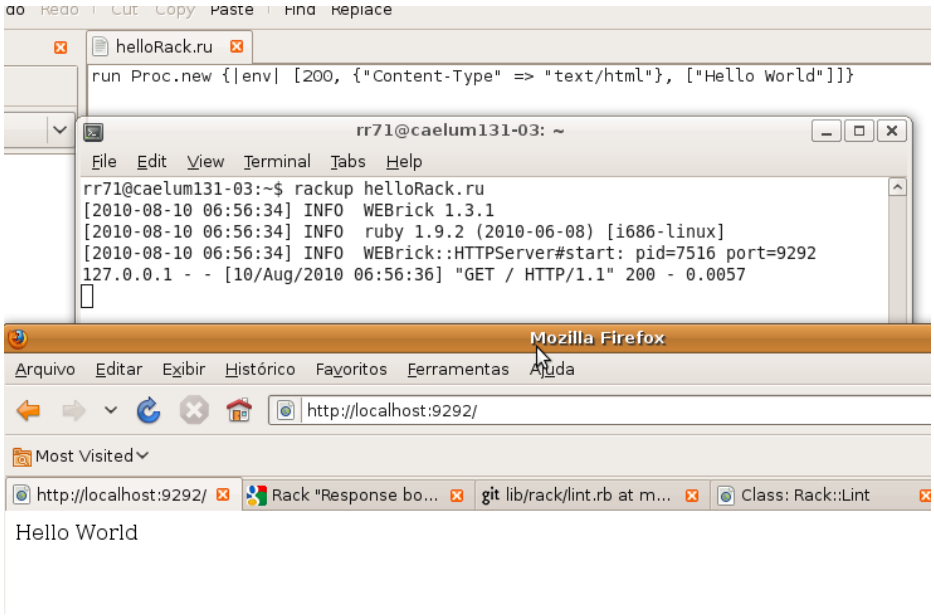
   a) Crie um arquivo chamado **"helloRack.ru"**

   b) Adicione as seguintes linhas:

   ```
   $ run Proc.new {|env| [200, {"Content-Type" => "text/html"},
     ["Hello World"]]}
   ```

   c) Inicie a aplicação com o comando `rackup helloRack.ru`

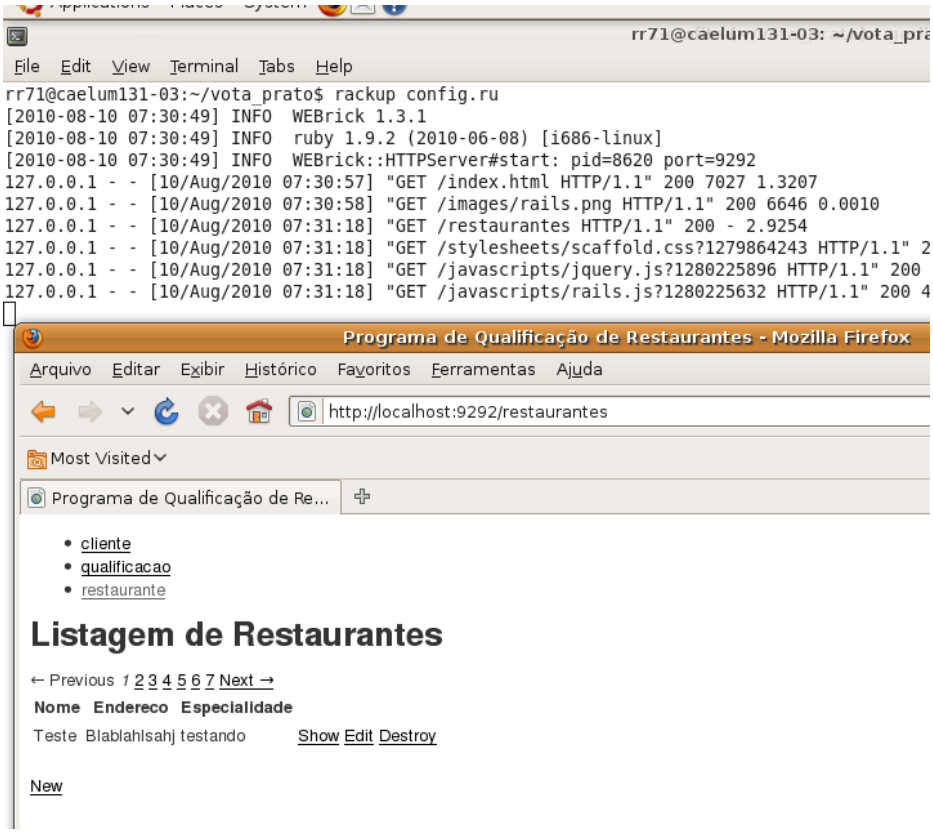
   d) Teste no browser pela url: http://localhost:9292/



17.3 Rails e o Rack

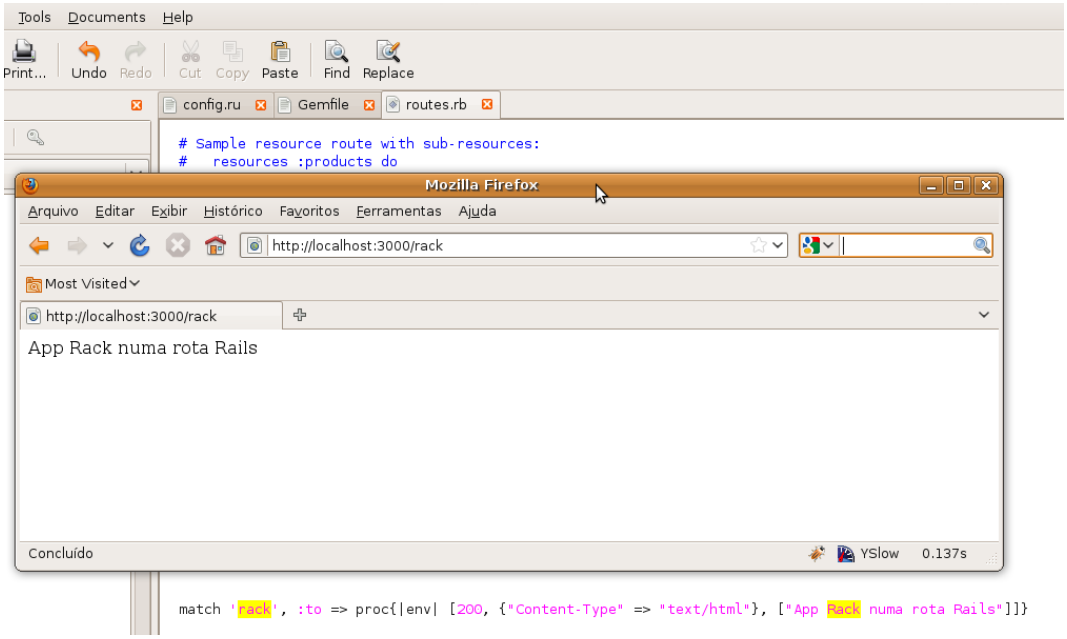
A partir do Rails 3, quando criamos uma nova aplicação, um dos arquivos que ele cria é config.ru na raiz do projeto e, mais que isso, podemos afirmar que toda aplicação Rails 3 é uma aplicação rack. Prova disso é que

conseguimos iniciar a aplicação através do comando `rackup config.ru`



Outro ponto interessante sobre o rack e o Rails, é que agora é possível mapear uma aplicação rack diretamente em uma rota de uma aplicação rails.

```
# routes.rb
  match 'rack',
  :to => proc{|env| [200, {"Content-Type" => "text/html"},
            ["App Rack numa rota Rails"]]}
```



17.4 Exercícios - Criando um rota para uma aplicação Rack

1) Vamos fazer uma aplicação rack.

   a) Abra o arquivo **"routes.rb"**

   b) Adicione a seguinte linha:

```
match 'rack',
 :to => proc{|env| [200, {"Content-Type" => "text/html"},
          ["App Rack numa rota Rails"]]}
```

   c) Inicie a aplicação com o comando `rails server`

   d) Teste no browser pela url: http://localhost:3000/rack

Capítulo 18

Apêndice: Design Patterns em Ruby

18.1 Singleton

Como criar uma classe com a garantia de ser instanciada apenas uma vez ? Essa classe deveria ser capaz de verificar se alguma vez já foi instanciada e saber devolver sempre a mesma referência.

Como visto anteriormente, em ruby, variáveis com @ são de instância e variáveis com @@ são variáveis de classe.

Utilizando a ideia acima podemos criar uma classe simples de relatório, onde desejamos que apenas uma seja de fato criada.

```
class Relatorio
    @@instance = Relatorio.new

    def self.instance
        return @@instance
    end
end
```

Dessa forma conseguimos criar o relatório apenas uma vez. Mas a implementação ainda apresenta problemas. Ainda é possível instanciar o Relatório mais de uma vez. Para resolver esse problema e implementar a classe Relatório como um Singleton realmente, precisamos tornar privado o **new** da nossa classe. Dessa forma, apenas será possível acessar o Relatorio a partir do método **instance**.

```
class Relatorio
    @@instance = Relatorio.new
```

```
    def self.instance
        return @@instance
    end

    private_class_method :new
end

# ambos relatórios referenciam o mesmo objeto
relatorio1 = Relatorio.instance
relatorio2 = Relatorio.instance
```

Existe ainda uma maneira mais otimizada e produtiva de chegar no mesmo resultado. O ruby já vem com um módulo chamado **Singleton**. Basta inclui-lo na classe para ter o mesmo resultado. Você verá mais sobre módulos no decorrer do curso.

```
require 'singleton'
class Relatorio
    include Singleton
end
```

18.2 Exercícios: Singleton

1) Crie o Relatório de forma a retornar sempre a mesma instância:

```
class Relatorio
    @@instance = Relatorio.new

    def self.instance
        return @@instance
    end

    private_class_method :new
end

# ambos relatórios referenciam o mesmo objeto
relatorio1 = Relatorio.instance
relatorio2 = Relatorio.instance

puts relatorio1 == relatorio2
```

2) Faça o mesmo teste, mas conhecendo agora o módulo Singleton do ruby:

```
require 'singleton'
class Relatorio
```

```
    include Singleton
end

# ambos relatórios referenciam o mesmo objeto
relatorio1 = Relatorio.instance
relatorio2 = Relatorio.instance

puts relatorio1 == relatorio2
```

18.3 Template Method

As operações de um sistema podem ser de diversos tipos e o que determina qual operação será realizada pode ser um simples variável. Dependendo da variável uma coisa pode ocorrer enquanto outra vez é necessário fazer outra. Para exemplificar melhor, vamos usar a classe **Relatorio**. Nosso relatório mostrará um conteúdo inicialmente em HTML.

```
class Relatorio
    def imprime
        puts "<html>Dados do restaurante</html>"
    end
end
```

O que acontece caso um parâmetro defina o formato dos dados que o relatório precisa ser criado?

```
class Relatorio
    def imprime(formato)
        if formato == :texto
            puts "*** Dados do restaurante ***"
        elsif formato == :html
            puts "<html>Dados do restaurante</html>"
        else
            puts "formato desconhecido!"
    end
end
```

Para solucionar esse problema de forma orientada a objetos poderíamos criar uma classe abstrata que define o comportamento de um relatório, ou seja, que define que um relatório de ter um head, um body, um footer, etc. Mas como criar uma classe abstrata em Ruby ? Embora não exista a palavra chave reservada “abstract” o conceito permanece presente na linguagem. Vejamos:

```
class Relatorio
    def imprime
        imprime_cabecalho
```

```
        imprime_conteudo
    end
end
```

A classe Relatorio agora possui os métodos que definem um relatório. Obviamente esses métodos podem ser invocados. A solução para isso normalmente é lançar uma exception nesses métodos não implementados. Agora que temos nossa classe abstrata, podemos criar subclasses de Relatorio que contém a implementação de cada um dos tipos, por exemplo para HTML:

```
class HTMLRelatorio < Relatorio
    def imprime_cabecalho
        puts "<html>"
    end

    def imprime_conteudo
        puts "Dados do relatorio"
    end
end

class TextoRelatorio < Relatorio
    def imprime_cabecalho
        puts "***"
    end

    def imprime_conteudo
        puts "Dados do relatorio"
    end
end
```

Agora para usar nossos relatórios podemos fazer:

```
relatorio = HTMLRelatorio.new
relatorio.imprime

relatorio = TextoRelatorio.new
relatorio.imprime
```

18.4 Exercícios: Template Method

1) Crie a classe que define as obrigações de um relatório:

```
class Relatorio
    def imprime
```

```
            imprime_cabecalho
            imprime_conteudo
        end
    end
```

2) Para termos implementações diferentes, crie um relatório HTML e um de texto puro:

```
class HTMLRelatorio < Relatorio
    def imprime_cabecalho
        puts "<html>"
    end

    def imprime_conteudo
        puts "Dados do relatorio"
    end
end

class TextoRelatorio < Relatorio
    def imprime_cabecalho
        puts "***"
    end

    def imprime_conteudo
        puts "Dados do relatorio"
    end
end
```

3) Utilize os relatórios com a chamada ao método imprime:

```
relatorio = HTMLRelatorio.new
relatorio.imprime

relatorio = TextoRelatorio.new
relatorio.imprime
```

18.5 Design Patterns: Observer

Como executar operações caso algo ocorra no sistema. Temos uma classe Restaurante e queremos manter avisados todos os objetos do sistema que se interessem em modificações nele. Nossa classe Franquia precisa executar o método **alerta** no caso

```
class Franquia

  def alerta
    puts "Um restaurante foi qualificado"
```

```
  end

end

class Restaurante

  def qualifica(nota)
    puts "Restaurante recebeu nota #{nota}"
  end

end

restaurante = Restaurante.new
restaurante.qualifica(10)
```

Poderíamos chamar o método **alerta** direto ao final do método **qualifica** ou poderíamos passar como parametro do método um observer para rodar. Porém essas maneiras acoplariam muito nosso código, por exemplo, não funcionariam para alertar mais de um observer.

Uma saída é criar uma lista de observadores e executá-los ao final da operação. Para isso podemos ter um método que permite adicionar quantos objetos forem necessários serem alertados.

```
class Restaurante

  def initialize
    @observers = []
  end

  def adiciona_observer(observer)
    @observer << observer
  end

  def notifica
    # percorre todos os observers chamando o método alerta
  end

  # qualifica

end

restaurante = Restaurante.new
restaurante.qualifica(10)
```

Poder ser observado não é uma característica única de um objeto. Podemos utilizar essa estratégia de avisar componentes por todo o software. Por que não colocar o comportamento em uma classe responsável por isso. Utilizando herança tornaríamos nosso código **observável**.

Ao invés de utilizar a herança novamente podemos utilizar os módulos para isso. Todo comportamento que faz do objeto ser um **observer** será colocado nesse módulo.

```
module Observer
    def initialize
       @observers = []
     end

     def adiciona_observer(observer)
       @observer << observer
     end

     def notifica
       # percorre todos os observers chamando o método alerta
     end
end

class Restaurante
    include Observer
  def qualifica(nota)
    puts "Restaurante recebeu nota #{nota}"
        notifica
  end
end
```

> **Modulo Observer**
>
> O módulo acima já existe em ruby e é chamado de Observer. http://ruby-doc.org/core/classes/Observable.html

18.6 Desafio: Design Pattern - Observer

1) Crie um novo arquivo e implemente o Design Pattern Observer para Restaurante e Franquia.

Capítulo 19

Apêndice: Integrando Java e Ruby

"Não há poder. Há um abuso do poder nada mais"
– Montherlant, Henri

Nesse capítulo você aprenderá a acessar código escrito anteriormente em Java através de scripts escritos em Ruby: o projeto JRuby.

19.1 O Projeto

JRuby (http://www.jruby.org/) é uma implementação de um interpretador Ruby escrito totalmente em java, e mais ainda, com total integração com a Virtual Machine.

Além de ser open-source, ele disponibiliza a integração entre as bibliotecas das duas linguagens.

Atualmente há algumas limitações como, por exemplo, não é possível herdar de uma classe abstrata. O suporte a Ruby on Rails também não está completo.

Os líderes desse projeto open source já trabalharam na Sun, o que permitiu termos uma implementação muito rápida e de boa qualidade.

19.2 Testando o JRuby

Vamos criar um script que imprima um simples "Testando o JRuby" na tela do seu console:

```
print "Testando o JRuby!\n"
```

Pode parecer muito simples, mas a grande diferença é que agora quem estará realmente rodando é uma Virtual Machine Java! Não há mais a necessidade de instalar o ruby na máquina, apenas a JVM e algumas bibliotecas do JRuby! Isto pode ajudar muito na adoção da linguagem.

Agora se executarmos tanto com os comandos "ruby" ou "jruby" o resultado será o mesmo:

19.3 Exercícios

1) Crie um arquivo chamado testando.rb que imprime na tela "Testando o JRuby!":

   a) Edite o arquivo: testando.rb.

   b) Adicione o seguinte conteúdo:

   ```
   print "Testando o JRuby!\n"
   ```

   c) Rode o arquivo com o JRuby:

   ```
   jruby testando.rb
   ```

19.4 Compilando ruby para .class com JRuby

Existe a possibilidade de compilarmos o arquivo .rb para um .class através do JRuby. Para isso devemos utilizar o jrubyc(JRuby Compiler) de modo muito semelhante ao javac:

```
jrubyc <path do arquivo .rb>
```

Vamos criar um arquivo ola_mundo_jruby.rb:

```
# ola_mundo_jruby.rb
puts 'Ola Mundo com JRuby!'
```

Agora vamos compilar esse arquivo:

```
jrubyc ola_mundo_jruby.rb
```

Após isso, o arquivo ola_mundo_jruby.class já foi criado na mesma pasta do arquivo ola_mundo_jruby.rb e nós podemos utilizá-lo a partir de outro arquivo .rb através do require, porém esse .class é diferente do que o javac cria a partir do .java, sendo assim é impossível rodá-lo direto na JVM como rodamos outra classe qualquer do java.

19.5 Rodando o .class do ruby na JVM

Como foi dito anteriormente, não é possível executar diretamente na JVM um arquivo compilado pelo jruby, isso acontece pelas características dinâmicas do ruby que tornam necessário a utilização de um jar. Tal jar pode ser baixada no site do Jruby(http://jruby.org/download).

Com o .jar em mãos, é fácil executar um bytecode do jruby na JVM, simplesmente devemos utilizar a opção "-jar" da seguinte maneira:

```
java -jar <path do arquivo .jar> <path do arquivo .class>
```

Lembrando que é necessário que a extensão do arquivo(.class) esteja explícita.

Vamos copiar o arquivo .jar do jruby para a pasta onde o ola_mundo_jruby.class está e rodar o nosso olá mundo:

```
java -jar jruby.jar ola_mundo_jruby.class
```

Após isso veremos nosso "Ola Mundo com JRuby!".

19.6 Importando um bytecode(.class) criado pelo jruby

Para importar um bytecode que foi criado a partir de um arquivo .rb utilizamos o conhecido require.

```
require '<path do bytecode>'
```

Obs.: Lembre-se de retirar a extensão do arquivo (.class), o certo seria fazer algo como:

```
require 'app/funcionario'
```

e não:

```
# desta maneira o arquivo não será encontrado
require 'app/funcionario.class'
```

19.7 Importando classes do Java para sua aplicação JRuby

Para importar classes Java utilizamos o método java_import, porém devemos ter o cuidado de antes requerer a biblioteca que tem esse método.

Vamos criar uma classe Pessoa em Java, e importar ela para dentro do JRuby. Primeiramente criaremos a classe Pessoa no java:

```
// Pessoa.java
        public class Pessoa {
            private String nome;

            public Pessoa(String meuNome) {
                this.nome = meuNome;
            }

            public void setNome(String novoNome) {
                this.nome = novoNome;
            }

            public String getNome(){
                return this.nome;
            }

            public void seMostra(){
                System.out.println(this.getNome());
            }
        }
```

Agora vamos compilar o código fonte com o javac utilizando:

```
javac Pessoa.java
```

Teremos então o arquivo Pessoa.class. Vamos criar um arquivo testando_jruby.rb onde vamos testar essa classe:

```
# testando_jruby.rb
        require 'java' # o java_import faz parte desta biblioteca
        java_import 'Pessoa'

        pessoa = Pessoa.new 'João'
        pessoa.se_mostra
        # Observe que o nome do método no código Java é
        # seMostra, porém o JRuby faz um alias para
        # todos os métodos passando-os de Camelcase para
        # Underscore case.
        # Obs.: o método seMostra ainda existe.

        pessoa.nome = 'Carlos'
        # Observe que ao criarmos um setter
        # para o nome(setNome), o JRuby criou
        # o método nome= automaticamente.
        # Obs.: Os métodos setNome e set_nome
```

```
        # continuam existindo.

        puts pessoa.nome
        # Observe que ao criarmos um getter
        # para o nome(getNome), o JRuby criou
        # o método nome automaticamente
        # Obs.: Os métodos getNome e get_nome
        # continuam existindo.
```

Ao executarmos o exemplo acima, teremos como saída:

```
João
Carlos
```

Lembrando que para executar este exemplo basta utilizar

```
jruby testando_jruby.rb
```

19.8 Testando o JRuby com Swing

Agora vamos integrar nosso "Testando o JRuby" com um pouco de Java, criando uma janela. Instanciamos um objeto Java em JRuby usando a notação:

```
require 'java'

module Swing
  include_package 'java.awt'
  include_package 'javax.swing'
end

module AwtEvent
  include_package 'java.awt.event'
end

# Reparem que não é necessário herdar nem
# implementar nada, apenas definir o metodo
# com o nome que o java exige (duck typing)
class ListenerDoBotao
  def action_performed(evento)
    Swing::JOptionPane.showMessageDialog(
        nil, "ActionListener feito em ruby")
  end
end
```

```
frame = Swing::JFrame.new
painel = Swing::JPanel.new
frame.add painel

label = Swing::JLabel.new
# label.setText("Testando o JRuby!")
label.text = "Testando o JRuby!"
painel.add label

botao = Swing::JButton.new 'clique aqui'
botao.add_action_listener ListenerDoBotao.new
painel.add botao

frame.pack
frame.set_size(400, 400)
frame.visible = true
```

O include_package é parecido com um import, e depois estamos criando uma instância de JFrame. Dessa mesma maneira você pode acessar qualquer outra classe da biblioteca do Java. Assim você tem toda a expressividade e poder do Ruby, somado a quantidade enorme de bibliotecas do Java.

Para saber mais: Suporte a closure com JRuby

O JRuby permite a passagem de blocos de código como argumento para métodos do java que recebem como parâmetro uma interface que define apenas um método, assim como o futuro suporte a closures prometido para o Java 8. No exemplo acima poderíamos ter passado o ActionListener para o botão sem necessidade de escrever uma classe só para isso, e nem mesmo seria preciso instanciar um objeto, fazendo desta forma:

```
botao.add_action_listener do |evento|
    Swing::JOptionPane.showMessageDialog(nil, "ActionListener em closure")
end
```

Capítulo 20

Apêndice: Deployment

"Há noites que eu não posso dormir de remorso por tudo o que eu deixei de cometer."
– Mario Quintana

Como construir ambientes de produção e deployment para aplicações Rails sempre foram alguns dos maiores desafios desta plataforma. Existem diversos detalhes a serem considerados e diversas opções disponíveis.

20.1 Webrick

A forma mais simples de executar aplicações rails é usar o servidor que vem embutido em todas estas aplicações: **Webrick**.

É um servidor web muito simples, escrito em Ruby, que pode ser iniciado através do arquivo **script/server**, dentro do projeto:

```
cd projetorails
rails server
```

Por padrão, o Webrick inicia na porta 3000, porém isto pode ser mudado com a opção `-p`:

```
rails server -p 3002
```

Por ser muito simples, **não é recomendado** o uso do webrick em produção.

20.2 CGI

Uma das primeiras alternativas de deployment para aplicações Rails foi o uso de servidores web famosos, como o Apache Httpd. Porém, como o httpd só serve conteúdo estático, precisar delegar as requisições dinâmicas para processos Ruby que rodam o Rails, através do protocolo CGI.

Durante muito tempo, esta foi inclusive uma das formas mais comuns de servir conteúdo dinâmico na internet, com linguagens como Perl, PHP, C, entre outras.

O grande problema no uso do CGI, é que o servidor Web inicia um novo processo Ruby a cada requisição que chega. Processos são recursos caros para o sistema operacional e iniciar um novo processo a cada requisição acaba limitando bastante o tempo de resposta das requisições.

20.3 FCGI - FastCGI

Para resolver o principal problema do CGI, surgiu o **FastCGI**. A grande diferença é que os processos que tratam requisições dinâmicas (*workers*) são iniciados junto ao processo principal do servidor Web.

Desta forma, não é mais necessário iniciar um novo processo a cada requisição, pois já foram iniciados. Os processos ficam disponíveis para todas as requisições, e cada nova requisição que chega usa um dos processos existentes.

> **Pool de processos**
>
> O conjunto de processos disponíveis para tratar requisições dinâmicas também é popularmente conhecido como **pool** do processos.

A implementação de FCGI para aplicações Rails, com o apache Httpd nunca foi satisfatória. Diversos bugs traziam muita instabilidade para as aplicações que optavam esta alternativa.

Infelizmente, FCGI nunca chegou a ser uma opção viável para aplicações Rails.

20.4 Lighttpd e Litespeed

Implementações parecidas com Fast CGI para outros servidores Web pareceram ser a solução para o problema de colocar aplicações Rails em produção. Duas alternativas ficaram famosas.

Uma delas é a implementação de Fast CGI e/ou SCGI do servidor web **Lighttpd**. É um servidor web escrito em C, bastante performático e muito leve. Muitos reportaram problemas de instabilidade ao usar o Lighttpd em aplicações com grandes cargas de requisições.

Litespeed é uma outra boa alternativa, usado por aplicações Rails em produção até hoje. Usa o protocolo proprietário conhecido como LSAPI. Por ser um produto pago, não foi amplamente difundido dentro da

comunidade de desenvolvedores Rails.

http://www.litespeedtech.com/ruby-lsapi-module.html

20.5 Mongrel

Paralelamente às alternativas que usam FCGI (e variações) através de servidores Web existentes, surgiu uma alternativa feita em Ruby para rodar aplicações Rails.

Mongrel é um servidor web escrito por Zed Shaw, em Ruby. É bastante performático e foi feito especificamente para servir aplicações Rails. Por esses motivos, ele rapidamente se tornou a principal alternativa para deployment destas aplicações. Hoje suporta outros tipos de aplicações web em Ruby.

20.6 Proxies Reversos

O problema com o Mongrel é que uma instância do Rails não pode servir mais de uma requisição ao mesmo tempo. Em outras palavras, o Rails não é thread-safe. Possui um lock que não permite a execução de seu código apenas por uma thread de cada vez.

Por causa disso, para cada requisição simultânea que precisamos tratar, é necessário um novo processo Mongrel. O problema é que cada Mongrel roda em uma porta diferente. Não podemos fazer os usuários terem de se preocupar em qual porta deverá ser feita a requisição.

Por isto, é comum adicionar um **balanceador de carga** na frente de todos os Mongrels. É o balanceador que recebe as requisições, geralmente na porta 80, e despacha para as instâncias de Mongrel.

Como todas as requisições passam pelo balanceador, ele pode manipular o conteúdo delas, por exemplo adicionando informações de cache nos cabeçalhos HTTP. Neste caso, quando faz mais do que apenas distribuir as requisições, o balanceador passa a ser conhecido como **Proxy Reverso**.

> **Reverso?**
>
> Proxy é o nó de rede por onde passam todas as conexões que saem. O nome Proxy Reverso vem da ideia de que todas as conexões **que entram** passam por ele.

O principal ponto negativo no uso de vários Mongrels é o processo de deployment. A cada nova versão, precisaríamos instalar a aplicação em cada um dos Mongrels e reiniciar todos eles.

Para facilitar o controle (*start*, *stop*, *restart*) de vários Mongrels simultaneamente, existe o projeto **mongrel_cluster**.

20.7 Phusion Passenger (mod_rails)

Ninh Bui, **Hongli Lai** e **Tinco Andringa** da empresa Phusion decidiram tentar novamente criar um módulo para rodar aplicações Rails usando o Apache Httpd.

Phusion **Passenger**, também conhecido como **mod_rails**, é um módulo para o Apache Httpd que adiciona suporte a aplicações Web escritas em Ruby. Uma de suas grandes vantagens é usar o protocolo **Rack** para enviar as requisições a processos Ruby.

Como o **Rack** foi criado especificamente para projetos Web em Ruby, praticamente todos os frameworks web Ruby suportam este protocolo, incluindo o Ruby on Rails, o Merb e o Sinatra. Só por serem baseados no protocolo Rack, são suportados pelo **Passenger**.

A outra grande vantagem do mod_rails é a facilidade de deployment. Uma vez que o módulo esteja instalado no Apache Httpd, bastam três linhas de configuração no arquivo **httpd.conf**:

```
<VirtualHost *:80>
    ServerName www.aplicacao.com.br
    DocumentRoot /webapps/aplicacoes/projetorails
</VirtualHost>
```

A partir daí, fazer deployment da aplicação Rails consiste apenas em copiar o código para a pasta configurada no Apache Httpd. O mod_rails detecta que é uma aplicação Rails automaticamente e cuida do resto.

A documentação do Passenger é uma ótima referência:

http://www.modrails.com/documentation/Users%20guide.html

20.8 Ruby Enterprise Edition

Além do trabalho no mod_rails, os desenvolvedores da Phusion fizeram algumas modificações importantes no interpretador MRI. As mudanças podem trazer redução de até 30% no uso de memória em aplicações Rails.

O *patch* principalmente visa modificar um pouco o comportamento do Garbage Collector, fazendo com que ele não modifique o espaço de memória que guarda o código do Rails. Desta forma, os sistemas operacionais modernos conseguem usar o mesmo código Rails carregado na memória para todos os processos. Esta técnica é conhecida como Copy on Write; suportada pela maioria dos sistemas operacionais modernos.

Outra mudança importante promovida pelos desenvolvedores da Phusion foi o uso de uma nova biblioteca para alocação de memória, **tcmalloc**, no lugar da original do sistema operacional. Esta biblioteca é uma criação do Google.

20.9 Exercícios: Deploy com Apache e Passenger

1) Abra o FileBrowser e copie o projeto **restaurantes** para o Desktop. o projeto está em /rr910/Aptana Studio Workspace/restaurantes

2) Abra o terminal e digite:

```
install-httpd
```

Feche e abra o terminal novamente Esse comando baixa httpd e compila na pasta /home/apache. No nosso caso=> /home/rr910/apache

3) Ainda no terminal entre no diretório do projeto e rode a migration para atualizar o banco em produção:

```
cd Desktop/restaurante
rake db:migrate:reset RAILS_ENV=production
```

4) Abra o arquivo de configuração do Apache:

```
gedit /home/rr910/apache/conf/httpd.conf
```

Altere a linha abaixo:

```
Listen 8080
```

Adicione a linha a baixo em qualquer lugar do arquivo:

```
Servername http://localhost:8080
```

5) Suba o Apache, no terminal rode:

```
apachectl start
```

Acesse http://localhost:8080/ no browser e confira se o Apache subiu, deve aparecer a mensagem **It works!**.

6) vamos instalar o Passenger. no terminal rode:

```
gem install passenger
passenger-install-apache2-module
```

Quando o instalador surgir no terminal, pressione **enter**.

No fim, copie a instrução semelhante a essa:

```
LoadModule passenger_module
      /home/rr910/.gem/ruby/1.8/gems/passenger-2.2.5/ext/apache2/
  mod_passenger.so
 PassengerRoot /home/rr910/.gem/ruby/1.8/gems/passenger-2.2.5

 PassengerRuby /usr/bin/ruby1.8
```

7) Abra o arquivo de configuração do Apache:

```
gedit /home/rr910/apache/conf/httpd.conf
```

Adicione essas linhas ao final do arquivo:

```
<VirtualHost *:8080>
     DocumentRoot /home/rr910/Desktop/restaurante/public

</VirtualHost>

<Directory />
   Options FollowSymLinks
   AllowOverride None
   Order allow,deny
   Allow from all
</Directory>
```

8) No terminal, de o restart no apache:

```
apachectl restart
```

Acesse **http://localhost:8080/restaurantes**. Nossa aplicação está rodando no Apache com Passenger!

Índice Remissivo

ActiveRecord, 79
after_filter, 145
around_filter, 146

BDD, 202
before_filter, 145
Behavior Driven Development, 202
Boolean, 20
Builder, 143

Classes abertas, 28
comments, 11
Comparable, 46
Copy on Write, 230

Duck Typing, 56

Enumerable, 46
ERB, 120

for, 21

Gems, 5
Gemstone, 8
gets, 13

HAML, 143

I/O, 50
if / elsif / case / when, 20

Maglev, 8
MetaProgramming, 60
Migrations, 82
mongrel_cluster, 229
MSpec, 8

nokogiri, 195

Observer, 218
open-uri, 195
Operações aritméticas, 18
Operadores Booleanos, 20
OR, 21

ORM, 79

Paginação com kaminari, 191
Palavras Reservadas do Ruby, 12
puts, print e p, 11

rack, 210
Rake, 80
Ranges, 19
Regexp, 21
respond_to, 142
RSpec, 202
Ruby, 4
Ruby Enterprise Edition, 9

Search Engine Optimization, 113
Symbol, 19
Syntax Sugar, 31

tcmalloc, 230
Template Method, 216
Testes, 197
Tipos numéricos, 17