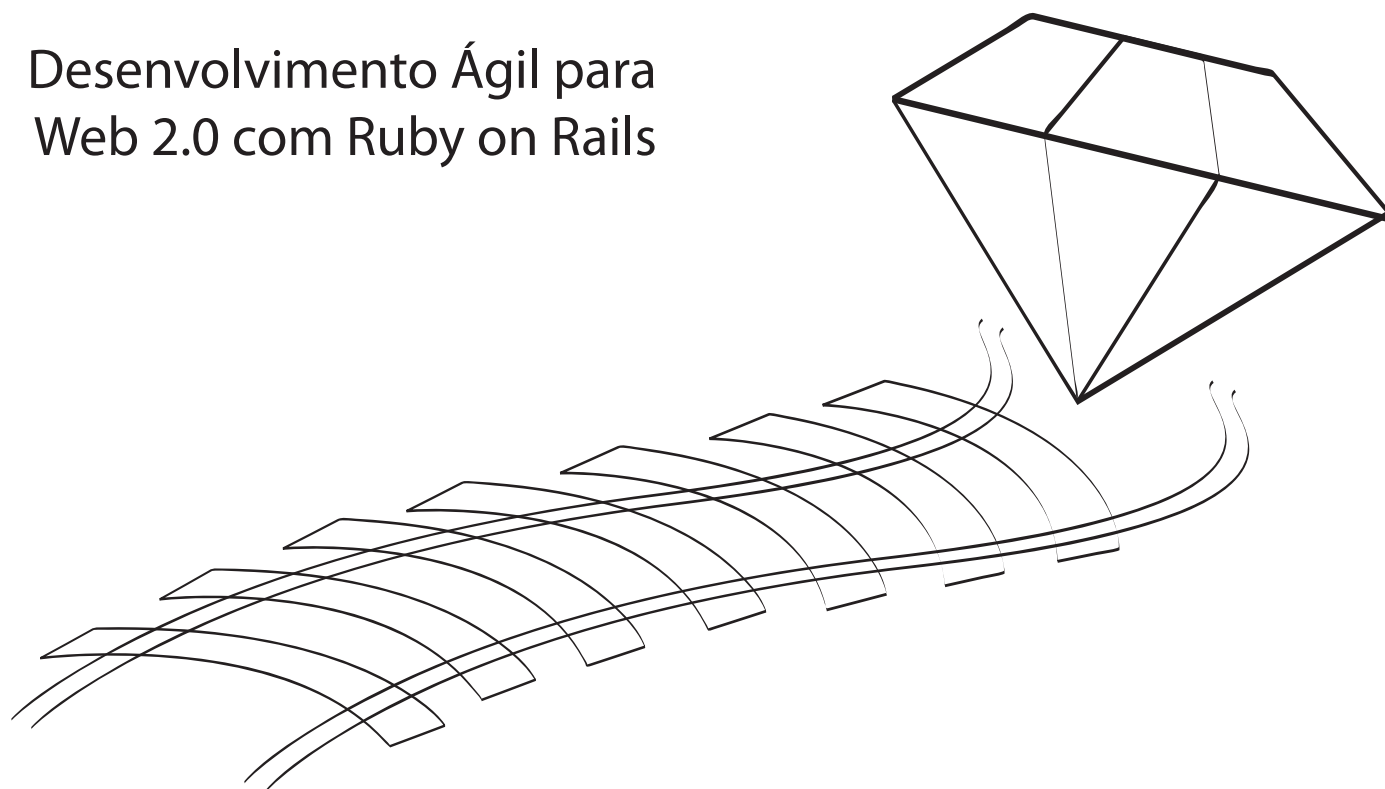


Caelum
Ensino e Inovação

RR-71

Desenvolvimento Ágil para
Web 2.0 com Ruby on Rails



www.caelum.com.br



A Caelum atua no mercado com consultoria, desenvolvimento e ensino em computação. Sua equipe participou do desenvolvimento de projetos em vários clientes e, após apresentar os cursos de verão de Java na Universidade de São Paulo, passou a oferecer treinamentos para o mercado. Toda a equipe tem uma forte presença na comunidade através de eventos, artigos em diversas revistas, participação em muitos projetos *open source* como o VRaptor e o Stella e atuação nos fóruns e listas de discussão como o GUJ.

Com uma equipe de mais de 60 profissionais altamente qualificados e de destaque do mercado, oferece treinamentos em Java, Ruby on Rails e Scrum em suas três unidades - São Paulo, Rio de Janeiro e Brasília. Mais de 8 mil alunos já buscaram qualificação nos treinamentos da Caelum tanto em nas unidades como nas próprias empresas com os cursos *incompany*.

O compromisso da Caelum é oferecer um treinamento de qualidade, com material constantemente atualizado, uma metodologia de ensino cuidadosamente desenvolvida e instrutores capacitados tecnicamente e didaticamente. E oferecer ainda serviços de consultoria ágil, mentoring e desenvolvimento de projetos sob medida para empresas.

Comunidade



Nossa equipe escreve constantemente artigos no **Blog da Caelum** que já conta com 150 artigos sobre vários assuntos de Java, Rails e computação em geral. Visite-nos e assine nosso RSS:

➡ blog.caelum.com.br



Acompanhe também a equipe Caelum no **Twitter**:

➡ twitter.com/caelumdev/equipe



O **GUJ** é maior fórum de Java em língua portuguesa, com 700 mil posts e 70 mil usuários. As pessoas da Caelum participam ativamente, participe também:

➡ www.guj.com.br



Assine também nossa **Newsletter** para receber as novidades e destaques dos eventos, artigos e promoções da Caelum:

➡ www.caelum.com.br/newsletter



No site da Caelum há algumas de nossas **Apostilas** disponíveis gratuitamente para download e alguns dos **artigos** de destaque que escrevemos:

➡ www.caelum.com.br/apostilas

➡ www.caelum.com.br/artigos

Conheça alguns de nossos cursos



FJ-11:
Java e Orientação a objetos



FJ-26:
Laboratório de MVC com
Hibernate e JSF para a Web



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



FJ-31:
Java EE avançado e
Web Services



FJ-19:
Preparatório para Certificação
de Programador Java



FJ-91:
Arquitetura e Design de
Projetos Java



FJ-21:
Java para Desenvolvimento
Web



FJ-27:
Spring Framework



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

Esse material é parte integrante do treinamento Desenvolvimento Ágil para Web 2.0 com Ruby on Rails e distribuído gratuitamente exclusivamente pelo site da Caelum. Todos os direitos são reservados à Caelum. A distribuição, cópia, revenda e utilização para ministrar treinamentos são absolutamente vedadas. Para uso comercial deste material, por favor, consulte a Caelum previamente.

www.caelum.com.br

Índice

1	Agilidade na Web	1
1.1	A agilidade	1
1.2	A comunidade Rails	1
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	4
2.2	Características	4
2.3	Instalação do interpretador	4
2.4	Outras implementações	6
2.5	MagLev	7
2.6	Ruby Enterprise Edition	7
2.7	Interactive Ruby	8
2.8	Tipos Básicos	8
2.9	Para Saber Mais - Desafios	9
2.10	Para Saber Mais - Desafio	10
3	Ruby Avançado	12
3.1	Mundo orientado a objetos	12
3.2	Métodos comuns	12
3.3	Meta-programação	12

3.4	Definição de métodos	13
3.5	Discussão: Enviando mensagens aos objetos	14
3.6	Classes	14
3.7	Desafio: Classes abertas	15
3.8	self	15
3.9	Desafio: self e o método puts	16
3.10	Atributos e propriedades: acessores e modificadores	16
3.11	Syntax Sugar	17
3.12	Métodos de Classe	18
3.13	Para saber mais: Singleton Classes	19
3.14	Metaprogramação	20
3.15	Convenções	22
3.16	Coleções	22
3.17	Blocos e Programação Funcional	23
3.18	Desafio: Usando blocos	25
3.19	Mais OO	25
3.20	Modulos	26
3.21	Manipulando erros e exceptions	27
3.22	Exercício: Manipulando exceptions	28
3.23	Arquivos Ruby	28
4	Ruby on Rails	30
4.1	Ruby On Rails - Apresentação	30
4.2	Aprender Ruby?	31
4.3	RadRails	31
4.4	Primeira Aplicação	33
4.5	Exercícios: Iniciando o Projeto	33
4.6	Estrutura dos diretórios	35
4.7	O Banco de Dados	36
4.8	Exercícios: Criando o banco de dados	37
4.9	A base da construção: scaffold (andaime)	37
4.10	Exercícios: Scaffold	38

4.11	Gerar as tabelas	40
4.12	Versão do Banco de Dados	41
4.13	Exercícios: Migrar tabela	41
4.14	Server	42
4.15	Documentação do Rails	43
4.16	Exercício Opcional: Utilizando a documentação	44
5	Active Record	46
5.1	Motivação	46
5.2	Exercícios: Controle de Restaurantes	46
5.3	Modelo - O “M” do MVC	48
5.4	ActiveRecord	48
5.5	Rake	49
5.6	Criando Modelos	50
5.7	Migrations	50
5.8	Exercícios: Criando os modelos	52
5.9	Manipulando nossos modelos pelo console	56
5.10	Exercícios: Manipulando registros	57
5.11	Exercícios Opcionais	59
5.12	Finders	59
5.13	Exercícios: Buscas dinâmicas	60
5.14	Validações	61
5.15	Exercícios: Validações	62
5.16	Exercícios - Completando nosso modelo	62
5.17	O Modelo Qualificação	65
5.18	Exercícios - Criando o Modelo de Qualificação	66
5.19	Relacionamentos	68
5.20	Para Saber Mais: Cache	70
5.21	Exercícios - Relacionamentos	70
5.22	Para Saber Mais - Eager Loading	74
5.23	Para Saber Mais - Named Scopes	74
5.24	Para Saber Mais - Modules	75

6	Controllers e Views	76
6.1	O “V” e o “C” do MVC	76
6.2	Hello World	76
6.3	Exercícios: Criando o controlador	77
6.4	Redirecionamento de Action e Action padrão	78
6.5	Trabalhando com a View: O ERB	79
6.6	Entendendo melhor o CRUD	80
6.7	Exercícios: Controlador do Restaurante	82
6.8	Helper	83
6.9	Exercícios: Utilizando helpers para criar as views	85
6.10	Partial	88
6.11	Exercícios: Customizando o cabeçalho	89
6.12	Layout	90
6.13	Exercícios: Criando o header	91
6.14	Outras formas de gerar a View	91
6.15	Filtros	92
7	Rotas	94
7.1	routes.rb	94
7.2	Pretty URLs	95
7.3	Named Routes	95
7.4	REST - map.resource	96
7.5	Actions extras em Resources	98
7.6	Diversas Representações	98
7.7	Para Saber Mais - Nested Resources	98
8	Completando o Sistema	100
8.1	Exercícios	100
8.2	Selecionando Clientes e Restaurante no form de Qualificações	103
8.3	Exercícios	103
8.4	Exercícios Opcionais	109

9 Calculations	110
9.1 Métodos	110
9.2 Média	110
9.3 Exercícios	111
10 Associações Polimórficas	112
10.1 Nosso problema	112
10.2 Alterando o banco de dados	112
10.3 Exercícios	114
11 Ajax fácil com RJS	117
11.1 Adicionando comentários nas views	117
11.2 Métodos de RJS Templates	117
11.3 Exercícios	119
11.4 Adicionando comentários	122
11.5 Exercícios	123
11.6 Exercícios - Enviando os dados com Ajax	126
12 Alguns Plugins e Gems Importantes	128
12.1 Paginação	128
12.2 Exercícios - Título	129
12.3 Hpricot	129
12.4 Exercícios - Testando o Hpricot	130
12.5 File Uploads: Paperclip	130
12.6 Exercícios	131
13 Apêndice A - Testes	133
13.1 O Porquê dos testes?	133
13.2 Test::Unit	133
13.3 RSpec	135
13.4 Cucumber, o novo Story Runner	138

14 Apêndice B - Integrando Java e Ruby	142
14.1 O Projeto	142
14.2 Testando o JRuby	142
14.3 Exercícios	142
14.4 Testando o JRuby com Swing	143
15 Apêndice C - Deployment	144
15.1 Webrick	144
15.2 CGI	144
15.3 FCGI - FastCGI	145
15.4 Lighttpd e Litespeed	145
15.5 Mongrel	145
15.6 Proxies Reversos	146
15.7 Phusion Passenger (mod_rails)	146
15.8 Ruby Enterprise Edition	147
15.9 Exercícios: Deploy com Apache e Passenger	147
16 Apêndice D - Instalação	150
16.1 Ruby - Ubuntu	150
16.2 Ruby - Windows	151
16.3 Rails	151
16.4 JDK	151
16.5 Aptana	151
16.6 Mongrel	152
16.7 MySQL	152
16.8 SVN	152

Versão: 11.9.18

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 descrever de seus sentidos que admitir o fato”

– Fiodór Dostoievski, 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.

O que possibilita toda essa simplicidade 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 meta programação, 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 garanta 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 Rails Summit, maior evento de Rails da América Latina, com presença dos maiores nomes nacionais e internacionais de Ruby on Rails.

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

- <http://akitaonrails.com/> - Fábio Akita



- <http://www.nomedojogo.com/> - Carlos Brando
- <http://yehudakatz.com/> - Yehuda Katz
- <http://blog.plataformatec.com.br/> - José Valim
- <http://railsenvy.com/> - Rails Envy
- <http://www.rubyinside.com.br/> - RubyInside Brasil
- <http://rubyflow.com/> - Rubyflow
- <http://blog.caelum.com.br/> - Blog da Caelum
- <http://caueguerra.com> - Cauê Guerra
- <http://fabiokung.com/> - Fabio Kung
- <http://andersonleite.com.br/> - Anderson Leite
- <http://guilhermesilveira.wordpress.com/> - Guilherme Silveira

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 ano de 2009:

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

1.3 - Bibliografia

- **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.

- **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.

A linguagem Ruby

“Rails is the killer app for Ruby.”

– Yukihiro Matsumoto, Criador da linguagem Ruby

Neste capítulo, conheceremos a poderosa linguagem de programação Ruby, base para completo entendimento do framework Ruby on Rails.

2.1 - A história do Ruby

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.

2.2 - Características

Uma de suas principais características é a expressividade que possui. Matz teve como objetivo desde o início que Ruby 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.

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).

2.3 - 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).

Este interpretador é conhecido como *Matz's Ruby Interpreter* (MRI), ou CRuby (já que é escrito em C) e é também considerado a implementação de referência para a linguagem Ruby.

A última versão estável é a **1.8.x**, mas já está disponível a versão 1.9 da linguagem, também conhecida como **YARV**, que já pode ser usada em produção apesar de não existirem ainda muitas bibliotecas compatíveis

(conhecidas no mundo de Ruby como *gems*) uma vez que essa é uma versão de transição até o Ruby 2.0 onde serão inclusas diversas mudanças e novas funcionalidades.

A maioria das distribuições Linux possuem o pacote de uma das última versões estáveis (em geral, 1.8.7) pronto para ser instalado. O exemplo mais comum de instalação é para o Ubuntu:

```
sudo apt-get install ruby1.8 ruby1.8-dev
```

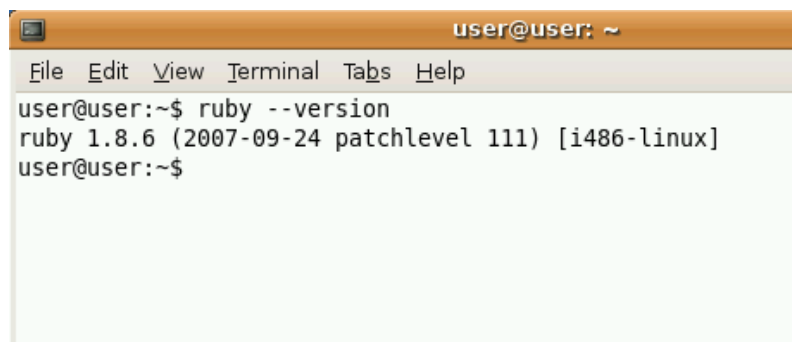
O interpretador ruby (*MRI*) já vem instalado no Mac OS X.

Apesar de existirem soluções prontas para a instalação do Ruby em diversas plataformas (*one-click-installers*), 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 1.8.6 (2007-09-24 patchlevel 111) [i486-linux]
```



A saída pode ser um pouco diferente, dependendo do sistema operacional e da versão instalada.

Ruby possui um gerenciador de pacotes e dependências bastante avançado, flexível e eficiente: **RubyGems**. Os *gems* são bibliotecas reutilizáveis de código Ruby, que podem até conter algum código nativo (em C, Java, .Net). São análogos aos jars do mundo Java, ou os assemblies do mundo .Net. RubyGems é um sistema gerenciador de pacotes comparável a qualquer um do mundo *NIX, como o `apt-get`, o MacPorts (BSD Ports), o `yum`, entre outros.

Portanto, para sermos capazes de instalar e utilizar as centenas de *gems* disponíveis, precisamos instalar além do interpretador Ruby, o Rubygems.

Basta baixá-lo através da url abaixo e executar o script de instalação contido no pacote:

<http://www.rubygems.org/>

```
cd rubygems-1.x.x
ruby setup.rb
```

2.4 - Outras implementações

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

O próprio Ruby 1.9 de referência (YARV), evolução do MRI, é baseado em uma máquina virtual: **Yet Another Ruby VM**.

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 foi a primeira implementação completa da versão 1.8.6 do Ruby. O JRuby é a principal implementação em Java da linguagem Ruby e é hoje considerada por muitos como a implementação mais rápida da linguagem.

Não é um simples interpretador, já que também opera nos modos AOT - compilação *Ahead Of Time* e JIT - Just In Time compilation, além do modo interpretador tradicional (*Tree Walker*).

Teremos um capítulo exclusivo sobre JRuby, mas 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 (GC, JIT, Threads nativas, entre outras).

Além disso, a própria Sun Microsystems aposta no projeto, já que alguns de seus principais desenvolvedores, Charles Nutter (líder técnico do projeto), Tomas Enebo e Nick Sieger já trabalharam para ela. Ola Bini também é contratado pela consultoria mundial ThoughtWorks para trabalhar apenas com JRuby.

O mundo .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.

Ruby.NET é outro projeto que tem como objetivo possibilitar código Ruby ser executado na plataforma .Net. Originalmente conhecido como *Gardens Point Ruby.NET Compiler*, procura ser um compilador de código Ruby para a *CLR* do mundo .Net.

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

<http://rubini.us>

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 unitários do projeto Rubinius, escritos com uma versão mínima do RSpec, conhecida como **MSpec**. O RSpec é um framework para descrição de especificações no estilo pregado pelo *Behavior Driven Development*. Veremos mais sobre isso no capítulo de testes.

2.5 - MagLev

Avi Bryant é um programador Ruby famoso, que mudou para Smalltalk e hoje é um defensor fervoroso da linguagem; além de ser um dos criadores do principal framework web em SmallTalk: **Seaside**. 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 demais. 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?

Algumas pessoas da empresa Gemstone, que possui uma das máquinas virtuais para SmallTalk mais famosas - Gemstone/S, estavam na platéia 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.

Apesar de ter feito bastante barulho durante a RailsConf 2008, a Gemstone anda bastante quieta sobre o **Maglev** e não mostrou mais nada desde então. Muitos criticam esta postura da empresa de ter falado sobre algo tão antes de poderem mostrar, além de terem exibido números que não podem provar.

Antonio Cangiano teve acesso a uma versão preliminar do **Maglev** e publicou um famoso comparativo de performance (*benchmark*) em seu blog, conhecido como *"The Great Ruby Shootout"*, em que o Maglev se mostra em média **1.8x mais rápido** que a MRI. Sendo muito mais rápido em alguns benchmarks e muito mais lento em alguns outros.

<http://antoniocangiano.com/2008/12/09/the-great-ruby-shootout-december-2008/>

2.6 - 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, ...).

Outra importante modificação foi na alocação de memória do interpretador, com o uso de bibliotecas famosas como `tcmalloc`.

Os desenvolvedores da Phusion já ofereceram as modificações (*patches*) para entrar na implementação oficial, MRI. É ainda um mistério para a comunidade o porquê de tais modificações importantes ainda não terem entrado para a versão oficial do interpretador.

http://izumi.plan99.net/blog/index.php/2008/08/17/_making-ruby's-garbage-collector-copy-on-write-friendly-part-8/

Copy on Write? tcmalloc?

Mais detalhes sobre estes assuntos estão no capítulo sobre Deployment.

No capítulo sobre **deployment**, veremos mais sobre o outro produto da empresa *Phusion*, o **Phusion Passenger** - também conhecido como **mod_rails**. Hoje, é o produto recomendado para rodar aplicações Rails em produção.

2.7 - Interactive Ruby

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.

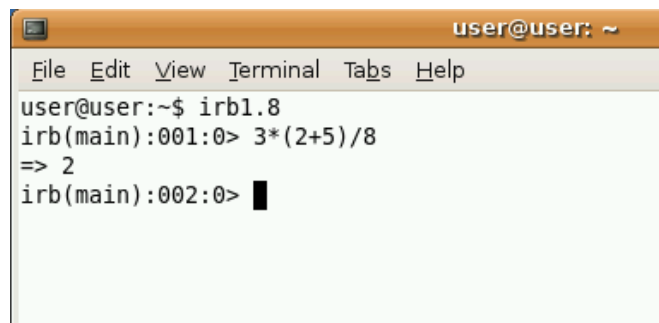
Todos os comandos apresentados neste capítulo podem ser executados dentro do `irb`, ou em arquivos `.rb`. Por exemplo, poderíamos criar um arquivo chamado **teste.rb** e executá-lo com o seguinte comando:

```
ruby teste.rb
```

2.8 - Tipos Básicos

Ruby permite avaliar expressões aritméticas tradicionais:

```
>> 3*(2+5)/8  
=> 2
```



Operadores Aritméticos

Estão disponíveis os operadores tradicionais: +, -, /, *, ** (potência) e % (resto da divisão inteira).

Os valores podem ainda ser atribuídos a variáveis:

```
>> resultado = 4 ** 2  
=> 16  
>> puts(resultado)  
16  
=> nil
```

`puts` é o método usado para imprimir valores na saída padrão (`stdout`) e `nil` é o objeto nulo do Ruby.

String

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

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

```
>> texto = "valor inicial"
=> "valor inicial"
>> texto << " alterado"
=> "valor inicial alterado"
>> puts(texto)
valor inicial alterado
=> nil
```

A concatenação de `Strings` (operador `+`) gera novas `Strings`, mesmo comportamento do Java. O operador `<<` é usado para a operação `append` de `Strings`.

Uma alternativa mais interessante para criar `Strings` com valor dinâmico é a interpolação:

```
n = 6 * 5
texto = "O resultado é #{n}. Algo maior seria #{n + 240}"
```

Qualquer expressão (código Ruby) pode ser interpolada em uma `String`. Porém, apenas `Strings` delimitadas por aspas duplas aceitam interpolação.

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

2.9 - Para Saber Mais - Desafios

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

```
n = 10 + 4
texto = "O valor é " + n
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?

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

4) Qual a saída deste código?

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

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

Symbol

Símbolos também são texto, como as Strings. Só que devem ser precedidos do caracter `':'`, 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. Ruby compartilha o mesmo objeto na memória em todos os lugares da aplicação que usarem um símbolo de mesmo valor. Todos os lugares da aplicação que contém o código `:coracao`, estão se referindo ao **mesmo objeto** na memória.

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 usado como identificadores ou nomes de alguma outra coisa.

2.10 - Para Saber Mais - Desafio

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

```
s1 = :abc
s2 = :abc
s1 == s2
# => true

t1 = "abc"
t2 = "abc"
t1 == t2
# => true
```

Boolean

Há também os objetos do tipo booleano, `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`:

```
>> abc = nil
=> nil
>> if(abc)
>>   puts("nao imprime")
>> end
=> nil
>> if(3 == 3)
>>   puts("agora imprime")
>> end
agora imprime
=> nil
```

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

```
>> /. *lalalala/ =~ "asdasdasdas"
=> nil
>> /sd.*/ =~ "asdasdasd"
=> 1
```

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.

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.

Ruby Avançado

“Rails is the killer app for Ruby.”

– Yukihiro Matsumoto, Criador da linguagem Ruby

3.1 - 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()
```

3.2 - Métodos comuns

Um 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 ou serialização. 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()
```

3.3 - Meta-programação

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

Imagine que tenho uma pessoa:

```
peessoa = Object.new()
```

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

```
peessoa.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**:

```
peessoa = Object.new()
```

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

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

```
peessoa = Object.new()
```

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

```
peessoa.fala()
```

Tudo isso é chamado **meta-programação**, um recurso muito comum de linguagens funcionais. Meta-programação é a capacidade de gerar 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.

3.4 - Definição de métodos

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

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

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

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

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

```
puts peessoa.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
```

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

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

3.5 - 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 idéia 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?

3.6 - 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
end
```



```
def troca(roupa, lugar="banheiro")
  "trocando de #{roupa} no #{lugar}"
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 coisas. Não será criada uma nova classe e nem haverá um erro dizendo que a classe já existe.

3.7 - 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 é útil?

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

3.8 - self

Um método pode chamar outro método do próprio objeto. Para isto, basta usar a referência especial `self`, que aponta 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)
  end
end
```



```
destino.deposita quantia
end
end
```

3.9 - 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 = "qualquer coisa"
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>

3.10 - 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 idéia 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"
```

3.11 - Syntax Sugar

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.

3.12 - 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
```

3.13 - 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 instancia 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 :

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

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

E onde estão os singleton methods ?

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 metodos desse objeto, sem compartilhar com outras instâncias da mesma classe.

E como defino uma singleton class ?

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
```

3.14 - Metaprogramação

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_accessor`, define os métodos `nome` e `nome=` na classe `Pessoa`.

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

Visibilidade dos métodos

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.

3.15 - 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.

3.16 - Coleções

Arrays em Ruby são instâncias da classe `Array`:

```
lista = Array.new
lista << "primeiro"
lista << "segundo"

puts lista[1]
# => "segundo"
```

Arrays literais podem ser criados com o uso de `[]` (brackets):

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

Todo objeto tem o método `methods`, que retorna um array com os nomes de todos os métodos que o objeto sabe responder.

Podemos declarar métodos que recebem uma quantidade arbitrária de argumentos através de um array:

```
def metodo(*args)
  puts args[0]
  puts args[1]
  # ...
end
```

```
metodo("abc", "xpto", "pele", "zzz")
```

O operador *, neste caso, é chamado de *splat*.

Os índices nem sempre precisam ser objetos `Fixnum` (inteiros). Ruby também tem uma estrutura indexada por qualquer objeto, onde as chaves podem ser objetos de qualquer tipo. Tais objetos em Ruby são instâncias da classe `Hash`, sendo análogos aos objetos `HashMap`, `HashTable`, arrays indexados por `String` e dicionários de outras linguagens.

```
p = Pessoa.new
mapa = Hash.new
mapa[:item1] = "lala"
mapa['item2'] = "lele"
mapa[p] = "lulu"
```

Por serem únicos e imutáveis, símbolos são ótimos candidatos a serem chaves em Hashes.

Uma técnica de programação em Ruby muito conhecida é o uso de Hashes como parâmetro único em métodos. A legibilidade do código aumenta já que 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:

```
def transfere(argumentos)
  destino = argumentos[:destino]
  valor = argumentos[:valor]
  # ...
end
```

```
transfere({ :destino => outra_conta, :valor => 100.0 })
```

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

```
transfere :destino => outra_conta, :valor => 100.0
```

3.17 - Blocos e Programação Funcional

Qualquer método em Ruby pode receber um pedaço arbitrário de código:

```
def meu_metodo
  puts "oi"
end
```

```
meu_metodo { puts "codigo associado" }
```

Este pedaço de código associado a chamada do método é conhecido como **bloco de código** (`Block`). Pode ter várias linhas, e nesse caso a preferência é pela sintaxe com `do` e `end` ao invés de chaves.

```
meu_metodo do
  puts "este"
  puts :bloco
  puts "tem"
  puts :varias
  puts "linhas"
end
```

O método que recebe o bloco pode decidir se deve ou não chamá-lo. Para chamar o bloco associado, usamos a palavra reservada `yield`.

```
def meu_metodo
  puts "chamando o bloco associado"
  yield
end
meu_metodo { puts "bloco chamado" }
```

```
# => chamando o bloco associado
# => bloco chamado
```

Blocos podem receber parâmetros:

```
def passa_parametro
  yield("algun", "texto") # parenteses opcionais
end

passa_parametro { |m1, m2| puts "#{m1}, #{m2}" } # ou

passa_parametro do |m1, m2|
  puts "m1: #{m1}"
  puts "m2: #{m2}"
end
```

Entender quando usar blocos é complicado no início. Você pode enxergá-los como uma oportunidade do método delegar parte da responsabilidade a quem o chama.

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. Blocos trazem uma característica funcional à linguagem Ruby. Dizer que estamos passando uma função (pedaço de código) como parâmetro a outra função é a mesma coisa que passar blocos na chamada de métodos.

Todo Array possui o método `each`, que chama o bloco de código associado para cada um dos seus itens, passando o item como parâmetro ao bloco:

```
lista = ["4", "um", "cinco", "bla"]
lista.each do |item|
  puts item
end
```

A construção acima não parece trazer muitos ganhos se comparada a forma tradicional e imperativa de iterar em um array (`for`). Podemos continuar com a idéia analisando o método `map` (ou `collect`), que coleta os retornos de todas as chamadas do bloco associado:

```
lista = ["quatro", "um", "cinco", "bla"]
novo = lista.map do |item|
  item.upcase
end

novo # => ["QUATRO", "UM", "CINCO", "BLA"]
```

Na programação imperativa tradicional precisaríamos de no mínimo mais duas linhas, para o array auxiliar e para ir adicionando os itens maiúsculos no novo array.

Os outros métodos do módulo `Enumerable` seguem a mesma idéia: `find`, `find_all`, `grep`, `sort`, `inject`. Não deixe de consultar a documentação!

3.18 - Desafio: Usando blocos

1) Queremos imprimir as 7 cores do arco-íris a partir do seguinte código:

```
ai = ArcoIris.new
ai.each do |x|
  puts x
end
```

Crie a sua classe **ArcoIris** que ao ser invocada mostra as 7 cores do arco-íris. Tente lembrar dos conceitos de blocos e programação funcional para resolver.

3.19 - Mais OO

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."

3.20 - 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
```

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

3.21 - 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 parâmetro 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 com argumento algo que não pode tratar.
- **ArgumentError** : Causada por número incorreto de argumentos.

3.22 - 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:"
n = gets.to_i

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

puts "100/#{n} 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 fala(idade)
  raise ArgumentError, "Idade negativa !!?" unless idade > 0
end

fala(-1)
```

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

```
class MinhaException < Exception
end

begin
  puts "Lançando MinhaException..."
  raise MinhaException
rescue MinhaException => e
  puts "Foi lançada a exception: #{e}"
end
```

3.23 - Arquivos Ruby

Todos os arquivos fonte, contendo código Ruby devem ter a extensão `.rb`. Além disso, você pode dividir o código fonte em diversos arquivos e pastas. Para carregar o código Ruby de outro arquivo, basta usar o método `require`:

```
require 'meu_outro_fonte'

puts ObjetoDefinidoFora.new.algum_metodo
```

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. Caminhos relativos ou absolutos podem ser usados para incluir arquivos em outros diretórios:

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

A constante \$:, ou \$LOAD_PATH contém diretórios do **Load Path**:

```
$:
# => ["/Library/Ruby/Site/1.8", ..., "."]
```

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

```
load 'meu_outro_arquivo'
load 'meu_outro_arquivo'
# executado duas vezes!
```

Ruby on Rails

“A libertação do desejo conduz à paz interior”

– Lao-Tsé

4.1 - Ruby On Rails - Apresentação

David Heinemeier Hansson criou o Ruby on Rails para usar em um de seus projetos na empresa *37signals*, o Basecamp. Desde então, passou a divulgar o código e incentivar o uso do mesmo.

O Rails foi criado pensando na agilidade e praticidade que ele proporcionaria na hora de escrever os aplicativos para Web.

Ágil x Rápido

Muitas pessoas confundem a diferença entre programação ágil e programação rápida. Programar com rapidez é escrever muito código difícil em pouco tempo. Programar com agilidade é escrever códigos simples, obtendo o mesmo resultado.

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

O primeiro conceito faz com que você não tenha que repetir código e sim modularizá-lo. Ou seja, você escreve trechos de código que serão reutilizáveis e evita o *Copy And Paste* de código.

Já o segundo, indica a configuração por exceção: ao utilizar os padrões oferecidos pelo Rails, você configura apenas o que estiver fora do padrão.

A arquitetura de desenvolvimento em Rails é baseada no conceito de separar tudo em três camadas (*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.

Meta-Framework

Rails na verdade não é um framework, mas sim um conjunto de frameworks, alguns dos quais discutiremos adiante:

- ActiveRecord
- ActionPack (incluindo ActionController e ActionView)
- ActiveSupport
- ActiveResource
- ActionMailer
- ActiveWebServices (antigo e não mais incluído por padrão)

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.com, Typo (blog open source) e o Spokeo (ferramenta de agrupamento de sites de relacionamentos).

Uma lista completa de sites usando RubyOnRails com sucesso pode ser encontrada nestes dois endereços:

<http://rails100.pbwiki.com>

<http://blog.obiefernandez.com/content/2008/03/big-name-compan.html>

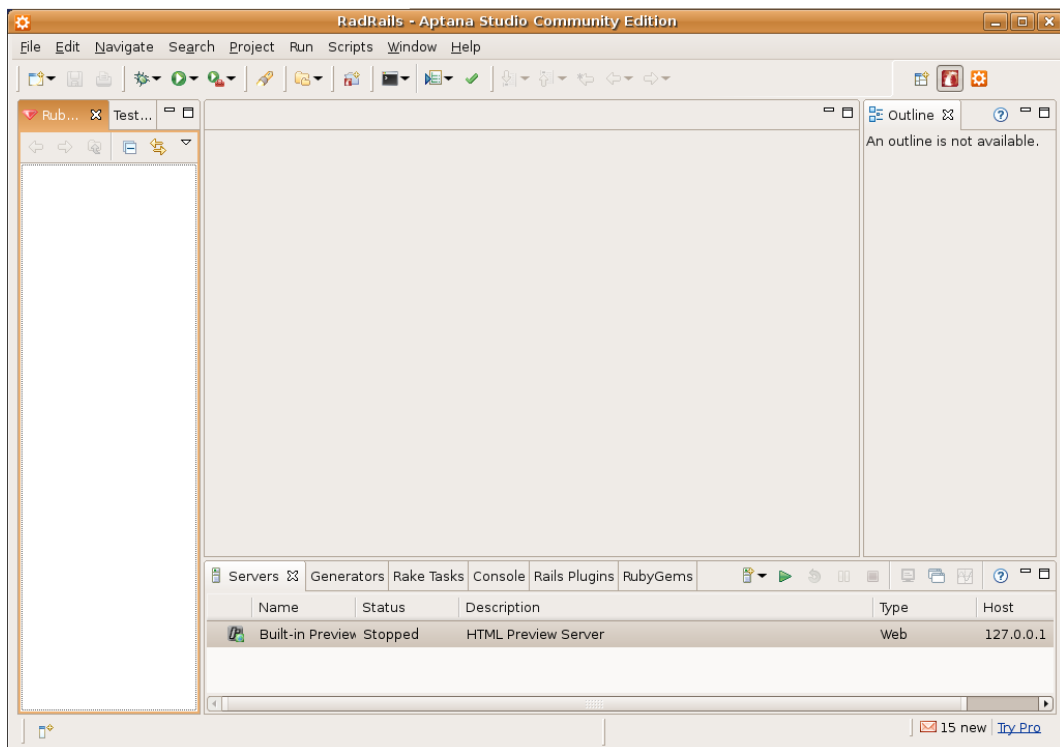
4.2 - Aprender Ruby?

Ruby on Rails é escrito em Ruby, mas você não precisa conhecer tal linguagem para começar a programar com ele. Tal necessidade surge com o tempo, à medida que precisamos utilizar recursos mais complexos.

A linguagem é dinâmica e interpretada, sendo que facilita o desenvolvimento das aplicações em alguns pontos básicos como, por exemplo, não obrigar o desenvolvedor a compilar o código ou a reiniciar um servidor para que a atualização feita entre no sistema.

4.3 - RadRails

O **Aptana RadRails** é um ambiente de desenvolvimento integrado (*Integrated Development Environment – IDE*) criado por Kyle Shank para facilitar ainda mais o desenvolvimento para o Ruby On Rails.



Existem diversas funções como execução de testes automatizados, suporte para debug, syntax highlighting, integração com diversas ferramentas, wizards, servidores, auto-complete, logs, perspectivas do banco de dados etc. O RadRails fornece tudo o que é preciso para desenvolver em Ruby on Rails: desde a criação do projeto até seu fechamento.

Outras IDEs

Existem diversas IDEs para trabalhar com Rails, sendo as mais famosas:

- NetBeans
- RubyMine (baseada no IntelliJ IDEA)
- RadRails
- MS Visual Studio 2008
- Ruby in Steel
- ScITE
- InType
- RoRed

Existem ainda muitos desenvolvedores que não usam IDE alguma. Apenas bons editores de texto, como o TextMate (Mac), Emacs, VI, GEdit, Kate, OpenKomodo, entre outros.

A instalação do RadRails é muito simples, basta você baixar o programa, descompactar e rodar. Para quem está acostumado a facilidade de instalação de programas escritos em Java, existe uma grande similaridade.

Para baixá-lo, visite o site oficial: <http://www.aptana.com/rails>

O RadRails é baseado no Eclipse RCP e hoje faz parte do projeto **Aptana** que possui várias ferramentas para desenvolvimento Web (Rails, PHP, JavaScript, Adobe Air, etc). O Aptana tem uma versão gratuita (Community Edition) e uma versão paga com algumas coisas a mais. A parte do RadRails é totalmente gratuita.

O RadRails precisa de uma definição sobre onde será ambiente de trabalho que será utilizado, ou seja, onde ficarão armazenados os projetos. Esse ambiente de trabalho é chamado Workspace e é o diretório base onde se localizam seus projetos.

Eclipse e RadRails

O RadRails é uma IDE baseada no Eclipse, que utiliza a JVM para rodar. Para executar o RadRails você deve instalar primeiro um Java Runtime Environment (JRE). Existe um plugin para o próprio eclipse que permite a programação em Ruby on Rails.

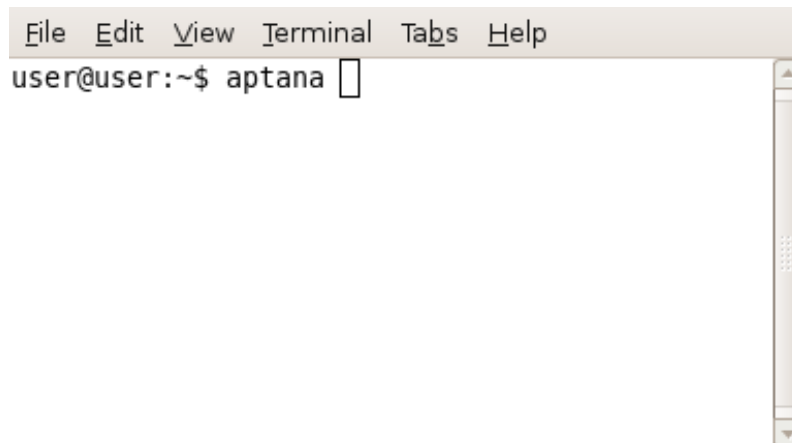
4.4 - Primeira Aplicação

Nesse capítulo, vamos montar uma pequena aplicação para apresentar algumas funcionalidades do RadRails. O sistema será uma agenda de indivíduos contendo nome, data de nascimento e altura.

4.5 - Exercícios: Iniciando o Projeto

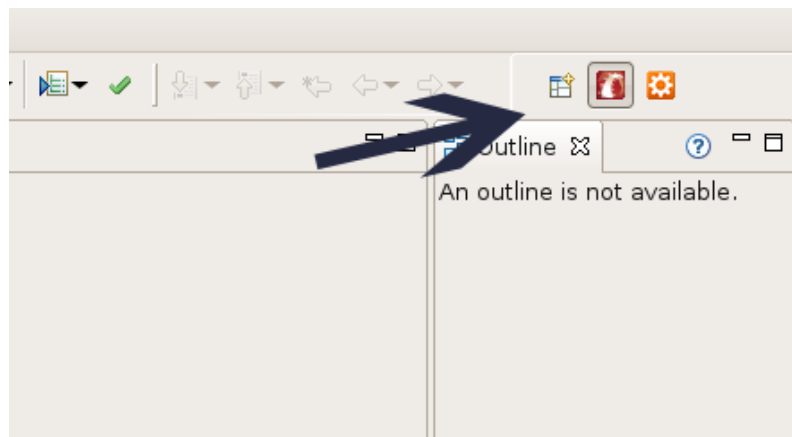
1) Inicie o RadRails:

- a) Abra o terminal
- b) Digite **aptana**

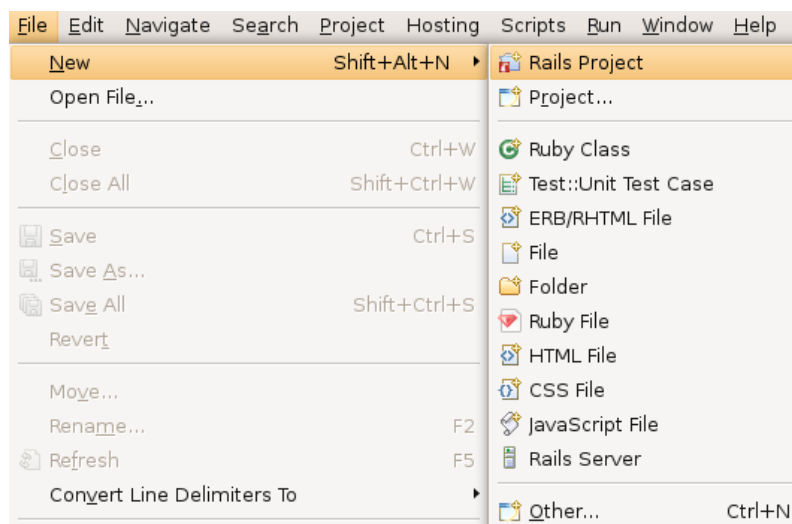


2) Inicie um novo projeto:

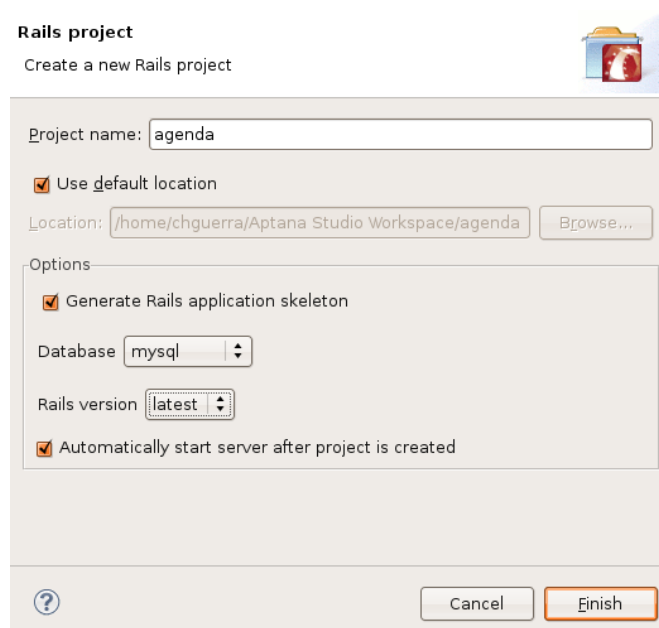
- a) Altere para a perspectiva "RadRails"



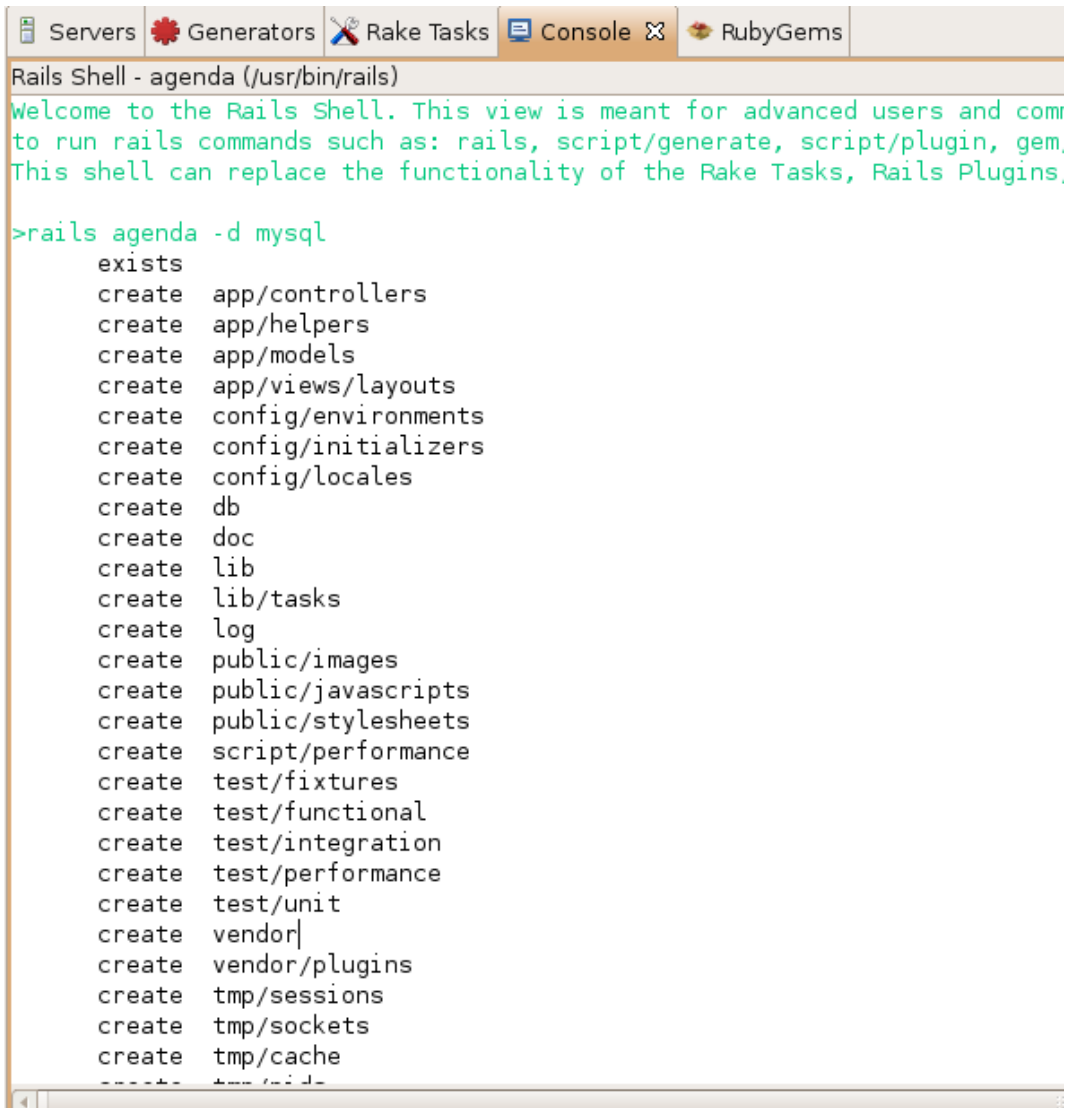
b) Clique em File -> New -> Rails Project.



c) Escolha o nome “agenda” para seu projeto, selecione **mysql** como base de dados e aperte Finish.



d) Verifique a view “console”. Note que o Rails já montou a estrutura do programa.



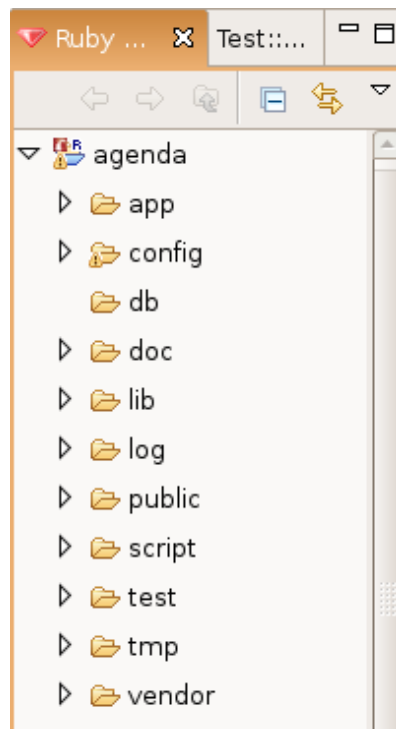
```
Servers Generators Rake Tasks Console RubyGems
Rails Shell - agenda (/usr/bin/rails)
Welcome to the Rails Shell. This view is meant for advanced users and comes
to run rails commands such as: rails, script/generate, script/plugin, gem.
This shell can replace the functionality of the Rake Tasks, Rails Plugins,

>rails agenda -d mysql
exists
create  app/controllers
create  app/helpers
create  app/models
create  app/views/layouts
create  config/environments
create  config/initializers
create  config/locales
create  db
create  doc
create  lib
create  lib/tasks
create  log
create  public/images
create  public/javascripts
create  public/stylesheets
create  script/performance
create  test/fixtures
create  test/functional
create  test/integration
create  test/performance
create  test/unit
create  vendor
create  vendor/plugins
create  tmp/sessions
create  tmp/sockets
create  tmp/cache
create  tmp/trace
```

4.6 - Estrutura dos diretórios

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

Ao manter selecionado no wizard de criação do projeto a opção “Generate Rails application skeleton”, o Rails gera toda a estrutura do seu programa, criando os seguintes diretórios.



- **app** -> 95% da aplicação e dos códigos criados ficam aqui (inclusive todo o MVC, dividido em diretórios).
- **config** -> Configurações da aplicação.
- **db** -> Banco de Dados, Migrações e Esquemas.
- **doc** -> Documentação do sistema.
- **lib** -> Bibliotecas.
- **log** -> Informações de log.
- **public** -> Arquivos públicos (CSS, imagens, etc), que serão servidos pela WEB.
- **script** -> Scripts pré-definidos do Rails (console, server).
- **test** -> Testes unitários, funcionais e de integração.
- **tmp** -> Qualquer coisa temporária como, por exemplo, cache e informações de sessões.
- **vendor** -> Plugins e programas externos.

4.7 - O Banco de Dados

Uma das características do Rails é a facilidade de se comunicar com diversos bancos 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 ao MySQL. O arquivo está localizado em **config/database.yml**.

Nada impede a alteração manual dessa configuração para outros tipos de bancos.

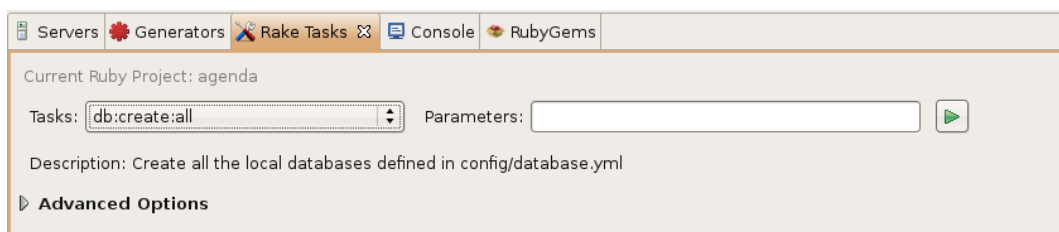
Nesse arquivo, configuramos três conexões diferentes: *development*, *test* e *production*, associados respectivamente a agenda_development, agenda_test e agenda_production.

Como esperado, *development* é usado na fase de desenvolvimento da aplicação e *production* em produção. A configuração para *test* se refere ao banco utilizado para os testes que serão executados, e deve ser mantido separado dos outros pois o framework apagará as tabelas após a execução.

Antes de começar a utilizar nosso banco de dados, devemos criá-los.

4.8 - Exercícios: Criando o banco de dados

- 1) Entre na view Rake Tasks e escolha a opção db:create:all para criar todas as tabelas no seu banco de dados.



- 2) Verifique que as três novas bases de dados agora foram criadas. Abra o terminal e logue como usuário root:

```
$ mysql -u root
```

```
mysql> show databases;
```

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| agenda_development |
| agenda_production  |
| agenda_test        |
| mysql              |
+-----+
5 rows in set (0.02 sec)

mysql> █
```

4.9 - A base da construção: scaffold (andaime)

Usando Rails, temos a opção de utilizar uma tarefa chamada **scaffold**, cuja tradução se equivale ao andaime de um prédio em construção: criar a base para a construção de sua aplicação.

Essa tarefa gera toda a parte de MVC (Model View Controller) relativa a algum modelo, conectando tudo o que é necessário para, por exemplo, lógicas de listagem, inserção, edição e busca, junto com suas respectivas visualizações.

Entre as partes do seu sistema que são geradas, existem os helpers, testes e o layout do site.

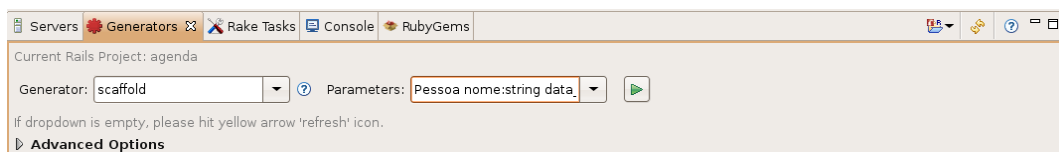
Repare também na estrutura de arquivos que é criada para o modelo de pessoas: o Rails coloca em seu devido lugar cada um dos controladores e visualizações.

No scaffold, existe também uma opção para passar como parâmetro o modelo a ser criado. Com isso, o Rails se encarrega de gerar o modelo e o script de criação da tabela no banco de dados para você.

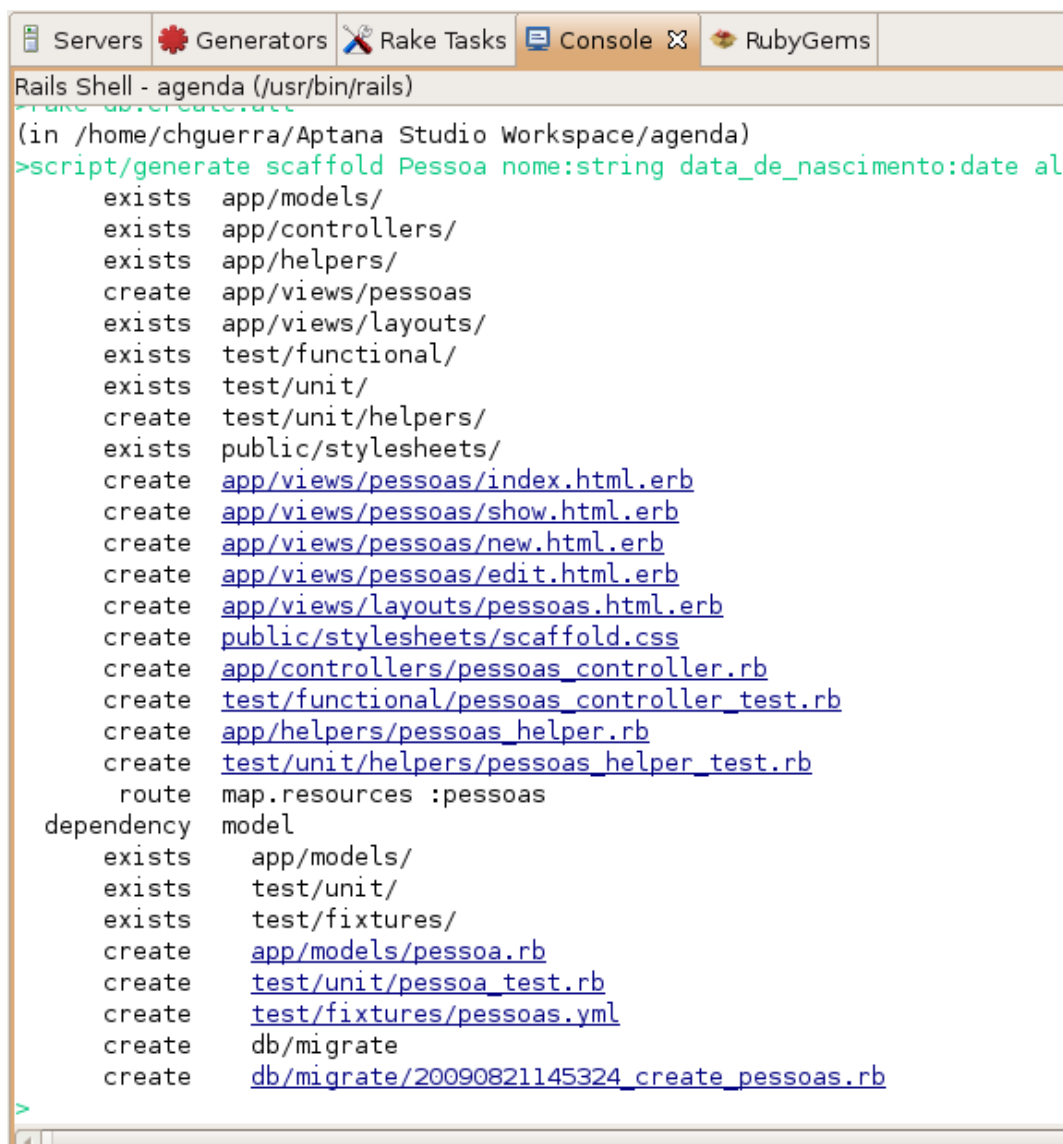
4.10 - Exercícios: Scaffold

1) Execute o scaffold do seu modelo pessoa:

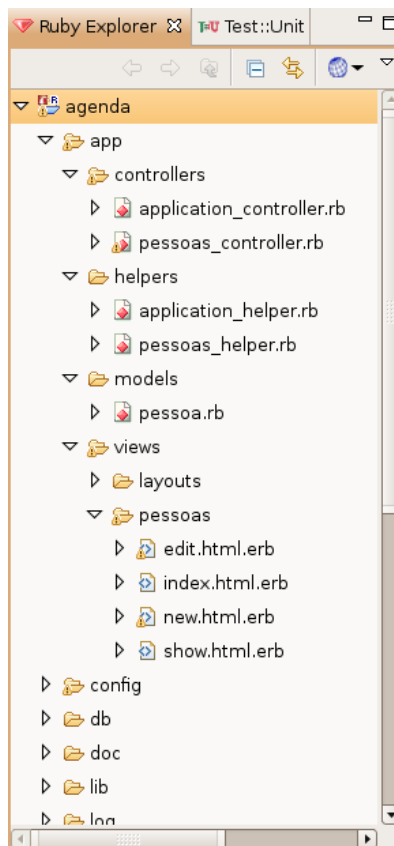
a) Entre na view Generators e escolha a opção scaffold;



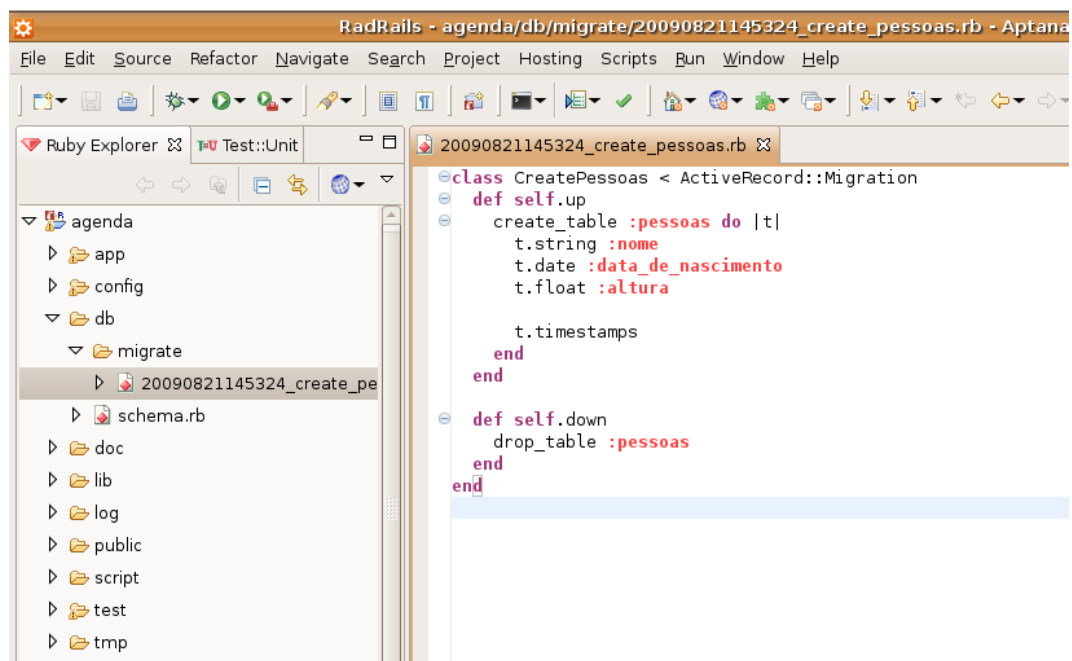
b) Digite "Pessoa nome:string data_de_nascimento:date altura:float"; Olhe no console os comandos que o Rails executa



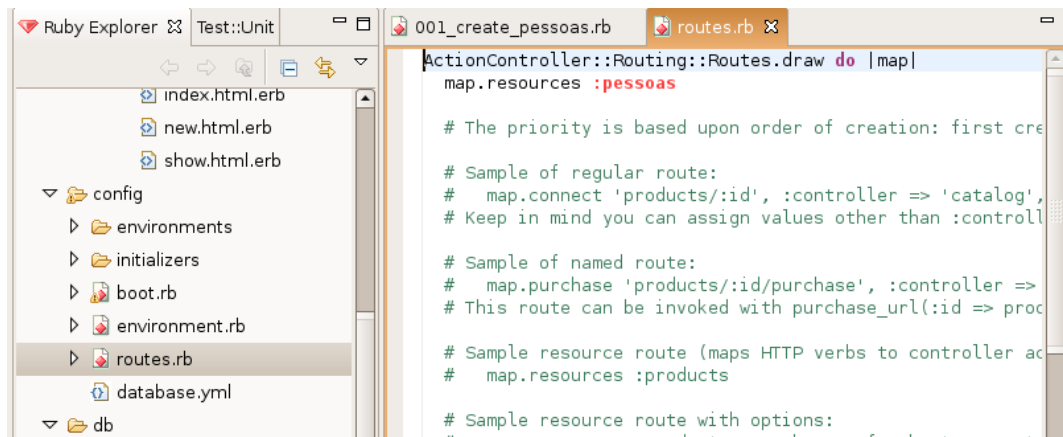
c) Além disso, diversos arquivos são criados



d) Também gera um arquivo para criação das tabelas no banco de dados



e) E um arquivo routes.rb, que veremos mais adiante



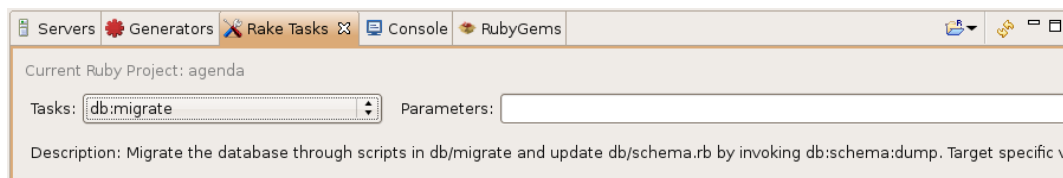
- f) (opcional) Criamos campos *string*, *date* e *float*. Quais outros tipos são suportados? Abra a documentação do Rails (api.rubyonrails.org) e procure pela classe **ActiveRecord::ConnectionAdapters::Table**.

4.11 - Gerar as tabelas

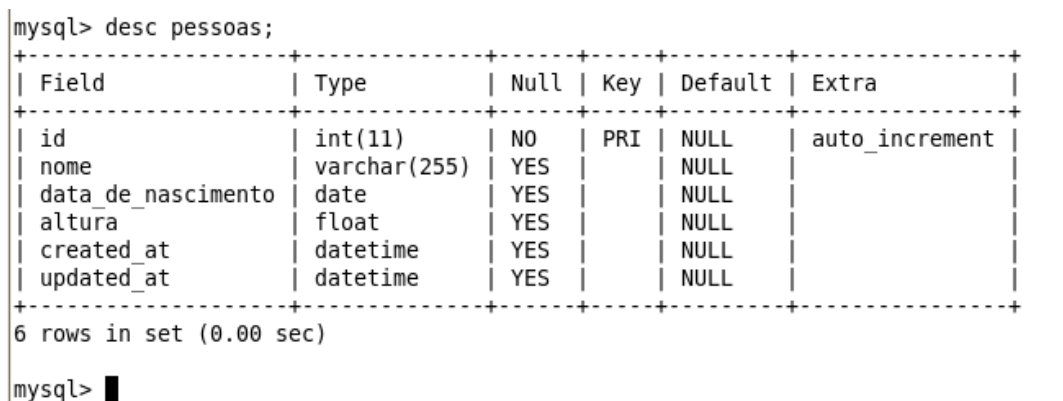
Já definimos o modelo e seus atributos, agora vamos executar o processo de geração das tabelas.

Basta chamar o comando `migrate` que o rails executa o script de migração gerado pelo scaffold.

O comando em questão é uma “**Rake Task**”. No próprio RadRails, existe uma view para execução desse tipo de comando, a “**Rake Tasks**”.



Repare também que os campos no MySQL correspondem exatamente ao que foi definido no código de migração.



O Rails gerou também, por conta própria, o campo “id”, fazendo o papel de chave primária de nossa tabela.

4.12 - Versão do Banco de Dados

Ao chamar a Rake Task “migrate” 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.

Versões anteriores

Nas versões anteriores do Rails, esta tabela se chamava **schema_info** e tinha apenas uma coluna, com exatamente uma linha. Ela servia para indicar qual a versão do esquema do banco de dados que utilizávamos. Esse número era o mesmo que o começo do nome do arquivo do script de migração. No nosso caso, como possuímos um único script, o nome começaria com 001 (001_create_pessoas.rb) e a versão do banco também seria 1.

Por exemplo, ao rodar a task de migração em um projeto que possui 20 scripts, cujo banco fora gerado ainda na versão 15, o Rails executaria os scripts 16, 17, 18, 19 e 20.

```
mysql> select * from schema_migrations;
+-----+
| version |
+-----+
| 20090821145324 |
+-----+
1 row in set (0.01 sec)

mysql>
```

Existe uma versão chamada 0, que retorna para o banco de dados em sua situação antes de ser atualizado pela primeira vez.

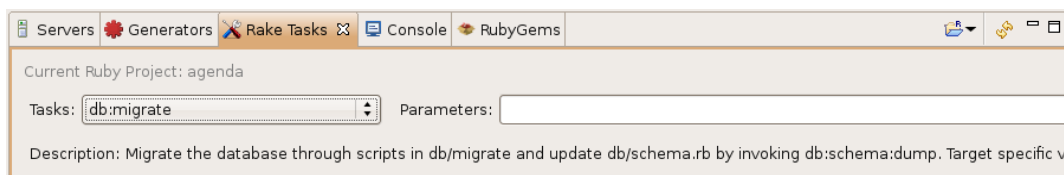
Através dessa tarefa, também podemos voltar o esquema do banco de dados para versões anteriores, bastando indicar nos argumentos da tarefa qual a versão que desejamos utilizar.

Quando voltamos a versões anteriores do esquema, o rake chama o método `down` do script, por isso é importante ter certeza que ele faz a operação inversa do método `up`.

4.13 - Exercícios: Migrar tabela

1) Migre a tabela para o banco de dados:

a) Entre na view “Rake Tasks”



b) Selecione a opção “db:migrate”

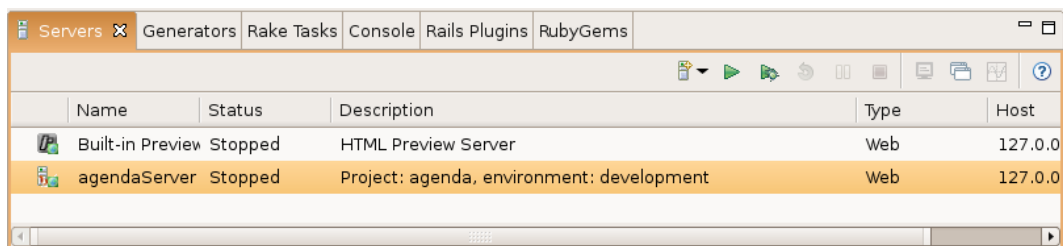
c) Aperte “go”

4.14 - Server

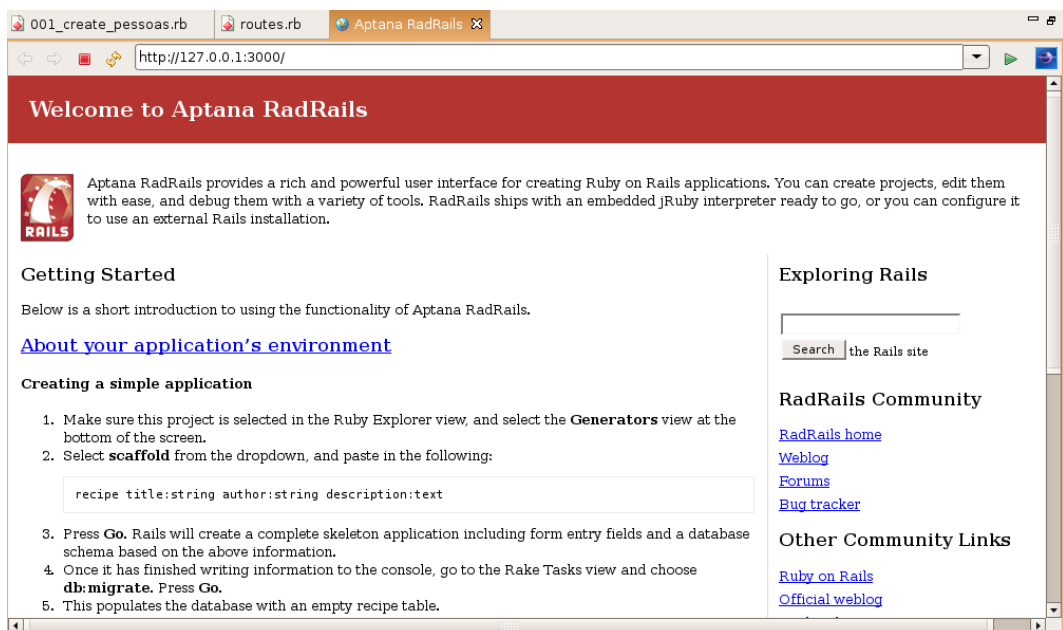
Precisamos de um servidor para rodar nossa aplicação web, mas ao criar a sua aplicação com o wizard, o RadRails já preparou um server, que, por padrão, usa a porta 3000.

Este servidor é capaz de interpretar scripts em Ruby que as páginas utilizem. Por padrão o Webrick é utilizado, sendo a opção mais simples. Mais detalhes no capítulo sobre deployment.

Podemos iniciar o servidor na view “Servers”, clicando no botão de start (seta verde para a direita).

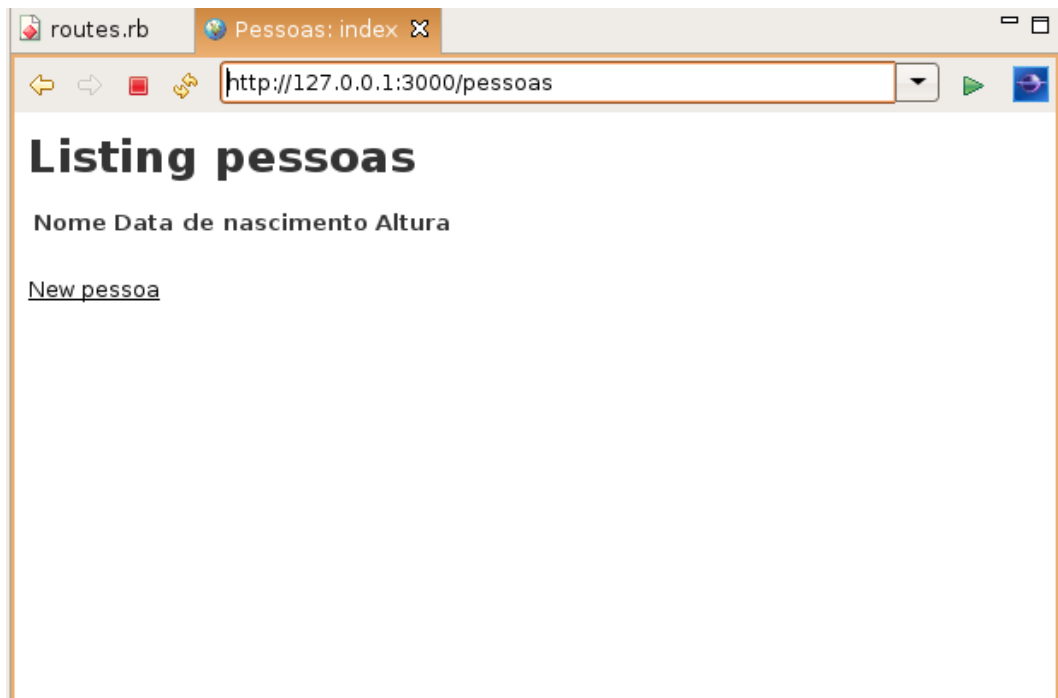


Após a inicialização do mesmo, clique com o botão direito em cima do server “agenda” e selecione *Launch Browser*, abrindo um browser dentro do próprio RadRails com o endereço `http://localhost:3000`



Ao gerar o scaffold, o Rails criou algumas lógicas e suas respectivas visualizações (views).

Por exemplo, ao visitar “`http://localhost:3000/pessoas`”, você é redirecionado para a listagem de pessoas.



4.15 - Documentação do Rails

O RDoc (documentação gerada a partir dos código fonte ruby) da versão corrente do Ruby on 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.noobkit.com>
- <http://www.railsmanual.com>
- <http://rails.raaum.org>
- <http://www.railsapi.org>
- <http://www.railsbrain.com>
- <http://www.gotapi.com>
- <http://delynnberry.com/projects/rails-chm-documentation>

O RubyGems também oferece a documentação para cada um dos gems que estão instalados, para isso basta iniciar o servidor de documentação embutido:

```
$ gem server
Starting gem server on http://localhost:8808/
```



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 doccom o rake:

```
rails docs
rake rails:freeze:gems
rake rails:doc
```

O truque é criar uma aplicação rails e congelar (*freeze*) o rails inteiro lá dentro. A operação *freeze* copia o código do rails para dentro da aplicação, na pasta `vendor/rails`.

Depois de executar a task `rails:doc`, 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 diversos excelentes *screencasts* (vídeos) com aulas e/ou palestras sobre diversos assuntos relacionados a Ruby e Rails. A lista está sempre crescendo, porém aqui estão alguns dos principais:

- <http://railscasts.com> - vídeos pequenos e de graça, normalmente de 15min a 45min. Mantidos por Ryan Bates.
- <http://peepcode.com> - vídeos maiores, verdadeiras vídeo-aulas; alguns chegam a 2h. São pagos, porém muitos consideram o valor simbólico (~\$9 por vídeo).
- <http://www.confreaks.com> - famosos por gravar diversas conferências sobre Ruby e Rails.
- <http://rubytu.be/> - reúne screencasts de diversas fontes.

4.16 - Exercício Opcional: Utilizando a documentação

- 1) Ao inserir uma nova pessoa, o campo **ano** apresentado no formulário mostra um ano muito atual. Como alterar ? Verifique no arquivo `app/views/pessoas/new.html.erb` que estamos usando um método chamado `date_select`:

```
<%= f.date_select :data_de_nascimento %>
```

Podemos procurar esse método na documentação, ele está em **ActionViewHelpersDateHelper** e uma das opções que ele fornece é:

- **:start_year** - Set the start year for the year select. Default is `Time.now.year - 5`.



Como utilizar essa informação? basta passar por hash, como abaixo:

```
<%= f.date_select :data_de_nascimento, :start_year => 1970 %>
```

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 do que os vistos anteriormente.

5.1 - Motivação

Queremos criar um sistema de qualificação de restaurantes.

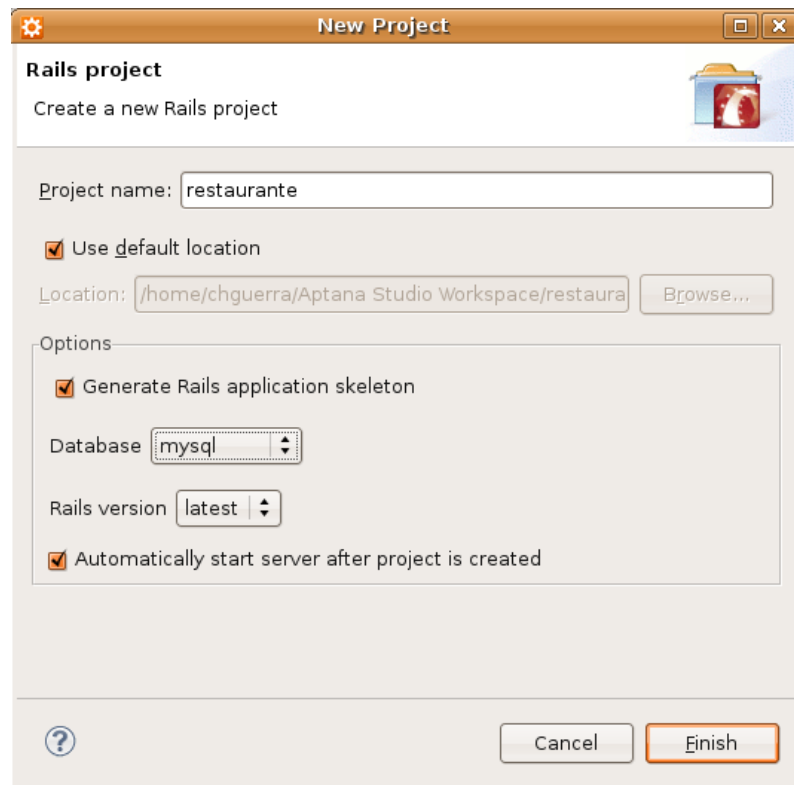
Esse sistema terá clientes que qualificam os restaurantes visitados com uma nota, além de informar quanto gastaram.

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.

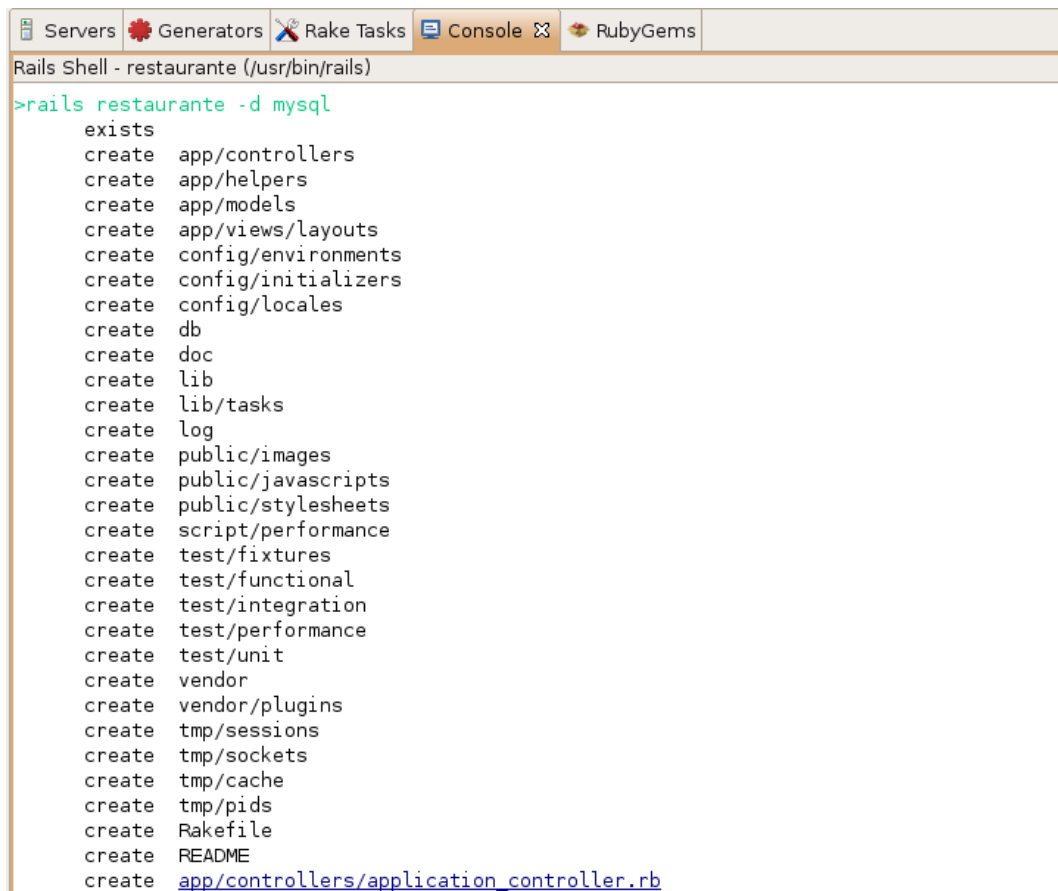
O site <http://www.tripadvisor.com> possui um sistema similar para viagens, onde cada cliente coloca comentários sobre hotéis e suas visitas feitas no mundo inteiro.

5.2 - Exercícios: Controle de Restaurantes

- 1) Crie um novo projeto chamado “restaurante”:
 - a) Clique no primeiro botão abaixo do menu e escolha “Rails Project”.
 - b) Digite “restaurante” para “Project Name”.



- c) Selecione **mysql** como banco de dados.
- d) Observe o log de criação do projeto:



```
Servers Generators Rake Tasks Console RubyGems
Rails Shell - restaurante (/usr/bin/rails)
>rails restaurante -d mysql
exists
create app/controllers
create app/helpers
create app/models
create app/views/layouts
create config/environments
create config/initializers
create config/locales
create db
create doc
create lib
create lib/tasks
create log
create public/images
create public/javascripts
create public/stylesheets
create script/performance
create test/fixtures
create test/functional
create test/integration
create test/performance
create test/unit
create vendor
create vendor/plugins
create tmp/sessions
create tmp/sockets
create tmp/cache
create tmp/pids
create Rakefile
create README
create app/controllers/application_controller.rb
```

5.3 - 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.

5.4 - 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.

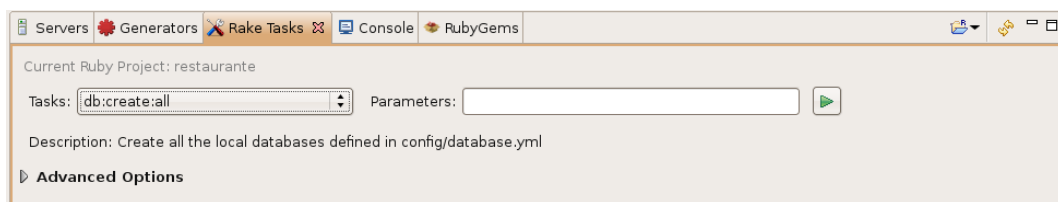
5.5 - Rake

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:all**. Para isso, vá para view *Rake*, escolha a opção **db:create:all** e clique em **go**.



Repare que o Rails cria três bancos de dados: **restaurante_development**, **restaurante_test** e **restaurante_production**.

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| agenda_development |
| agenda_production |
| agenda_test |
| mysql |
+-----+
5 rows in set (0.02 sec)

mysql>
```

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

```
rake -T
```

```

user@user: ~/Desktop/aptana/workspace
File Edit View Terminal Tabs Help
user@user:~/Desktop/aptana/workspace/restaurante$ rake -T
(in /home/user/Desktop/aptana/workspace/restaurante)
rake db:abort_if_pending_migrations # Raises an error if there are pending...
rake db:charset                     # Retrieves the charset for the curren...
rake db:collation                   # Retrieves the collation for the curr...
rake db:create                      # Create the database defined in confi...
rake db:create:all                  # Create all the local databases defin...
rake db:drop                       # Drops the database for the current R...
rake db:drop:all                   # Drops all the local databases define...
rake db:fixtures:identify          # Search for a fixture given a LABEL o...
rake db:fixtures:load              # Load fixtures into the current envir...
rake db:migrate                    # Migrate the database through scripts...
rake db:migrate:down               # Runs the "down" for a given migratio...
rake db:migrate:redo               # Rollbacks the database one migration...
rake db:migrate:reset              # Resets your database using your migr...
rake db:migrate:up                 # Runs the "up" for a given migration ...
rake db:reset                      # Drops and recreates the database fro...
rake db:rollback                   # Rolls the schema back to the previou...
rake db:schema:dump                # Create a db/schema.rb file that can ...
rake db:schema:load                # Load a schema.rb file into the database
rake db:sessions:clear             # Clear the sessions table
rake db:sessions:create            # Creates a sessions migration for use...
rake db:structure:dump             # Dump the database structure to a SQL...
rake db:test:clone                 # Recreate the test database from the ...
rake db:test:clone:structure       # Recreate the test databases from the...
rake db:test:prepare               # Prepare the test database and load t...
rake db:test:purge                 # Fmntv the test database
  
```

5.6 - Criando Modelos

Agora iremos criar o modelo do Restaurante. Para isso, entre na View **Generators**, selecione a opção **model** e dê o nome **restaurante** para o modelo.



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

```

>script/generate model restaurante
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/restaurante.rb
create test/unit/restaurante_test.rb
create test/fixtures/restaurantes.yml
create db/migrate
create db/migrate/20080722103736_create_restaurantes.rb
  
```

5.7 - 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

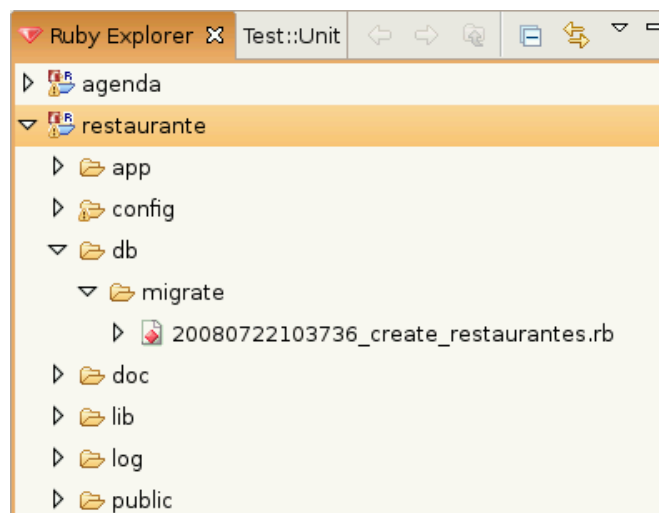
mudança para os demais desenvolvedores de um projeto e para o servidor de produção.

Com as migrations, podemos descrever essas transformações em classes que podem ser controladas por sistemas de controle de versão (por exemplo, SVN) 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.



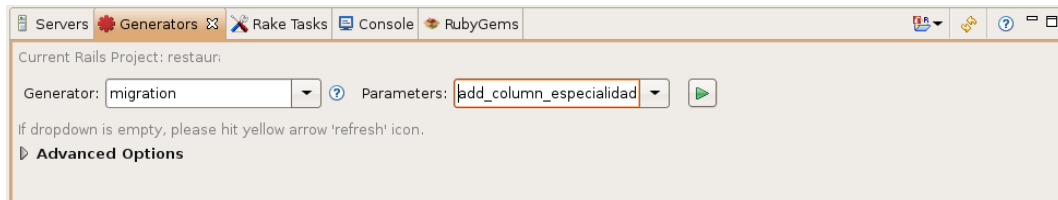
Queremos que nosso restaurante tenha um nome e um endereço. Para isso, devemos acrescentar:

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

Sua migration deve ter ficado parecida com esta:



Supondo que agora lembramos de adicionar a especialidade do restaurante. Como fazer? Basta entrar na view *Generator*, escolher a opção migration e dar um nome, por exemplo: **add_column_especialidade_to_restaurante**.



Como as migrations são sempre incrementais, basta incluirmos a coluna faltante:

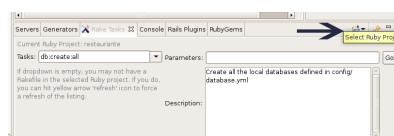
```
def self.up
  add_column :restaurantes, :especialidade, :string, :limit => 40
end

def self.down
  remove_column :restaurantes, :especialidade
end
```

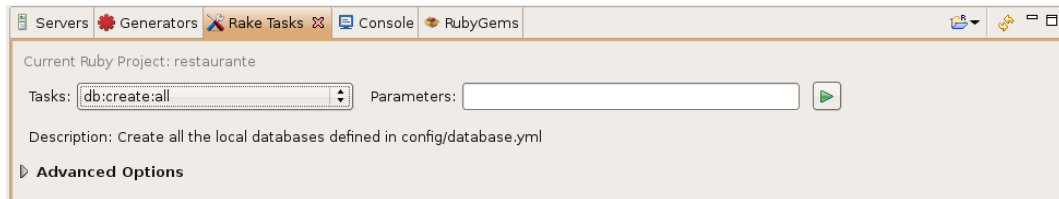


5.8 - Exercícios: Criando os modelos

- 1) Crie nosso banco de dados
 - a) Entre na view **Rake Tasks**
 - b) Altere para o projeto restaurante no ícone de projetos.



- c) Escolha a opção **“db:create:all”**
- d) Clique em **“go”**



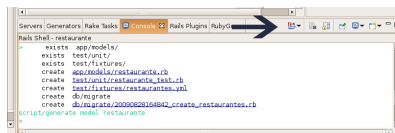
Entre no mysql e digite show databases;

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| agenda_development |
| agenda_production |
| agenda_test |
| mysql |
+-----+
5 rows in set (0.02 sec)

mysql>
```

2) Crie o modelo do Restaurante

- Entre na view “**Generator**”
- Altere para o projeto restaurante no ícone de projetos.



- Escolha a opção “**model**”
- Digite o nome “**restaurante**”
- Clique em “**go**”



Olhe no console o que foi feito e os diretórios criados.

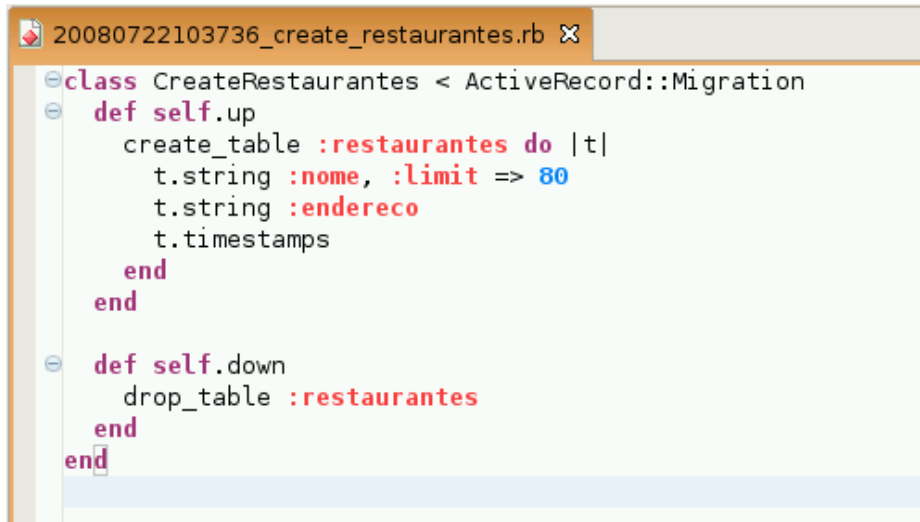
```
>script/generate model restaurante
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/restaurante.rb
create test/unit/restaurante_test.rb
create test/fixtures/restaurantes.yml
create db/migrate
create db/migrate/20080722103736_create_restaurantes.rb
```

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
```



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

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

4) Migre as tabelas para o banco de dados:

a) Abra a view **Rake Tasks**

b) Escolha a opção “db:migrate”

c) Aperte “go”

d) Olhe no console o que foi feito

```
>(in /home/user/Desktop/aptana/workspace/restaurante)
rake db:migrate
== 20080722103736 CreateRestaurantes: migrating =====
-- create_table(:restaurantes)
-> 0.0084s
== 20080722103736 CreateRestaurantes: migrated (0.0088s) =====
```

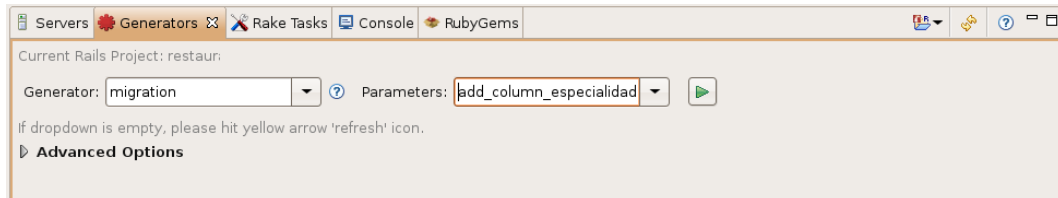
e) Olhe no banco de dados

```
mysql -u root
use restaurante_development;
desc restaurantes;
```

```
mysql> desc restaurantes;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id     | int(11) | NO | PRI | NULL | auto_increment |
| nome   | varchar(80) | YES | | NULL | |
| endereco | varchar(255) | YES | | NULL | |
| created_at | datetime | YES | | NULL | |
| updated_at | datetime | YES | | NULL | |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.01 sec)
```


5) Adicione a coluna especialidade ao nosso modelo “restaurante”:

- Abra a view “Generator”
- Escolha a opção “**migration**”
- De o nome “add_column_especialidade_restaurante”

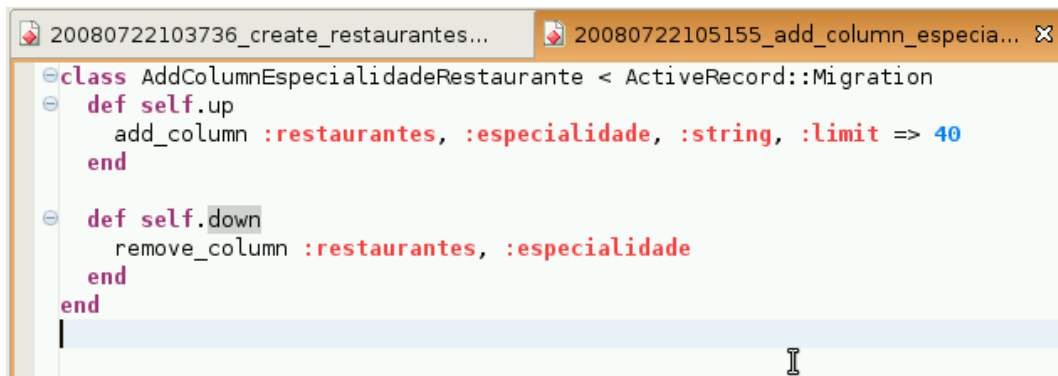


- Aperte “go”
- Abra o arquivo “db/migrate/<timestamp>_add_column_especialidade_restaurante.rb”
- Adicione as linhas:

```
def self.up
  add_column :restaurantes, :especialidade, :string, :limit => 40
end

def self.down
  remove_column :restaurantes, :especialidade
end
```

Seu arquivo deve ter ficado assim:



- Abra a view “Rake Tasks”
- Escolha a opção “**db:migrate**”
- Aperte “go”
- Olhe no console o que foi feito

```
>(in /home/user/Desktop/aptana/workspace/restaurante)
rake db:migrate
== 20080722105155 AddColumnEspecialidadeRestaurante: migrating =====
-- add_column(:restaurantes, :especialidade, :string, {:limit=>40})
   -> 0.0826s
== 20080722105155 AddColumnEspecialidadeRestaurante: migrated (0.0828s) =====
```

k) Olhe no banco de dados

```
mysql> desc restaurantes;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
nome	varchar(80)	YES		NULL	
endereco	varchar(255)	YES		NULL	
created_at	datetime	YES		NULL	
updated_at	datetime	YES		NULL	
especialidade	varchar(40)	YES		NULL	

```
6 rows in set (0.00 sec)
```

l) Utilizamos **add_column** e **remove_column** na nossa *migration* para adicionar uma nova coluna. O que mais poderia ser feito ? Abra a documentação e procure pelo módulo **ActiveRecordConnectionAdaptersSchemaStatements**.

5.9 - 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

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

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

Repare que o create já salva o novo restaurante no banco de dados, não sendo necessário o comando save.

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 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.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 obedecem 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()`:

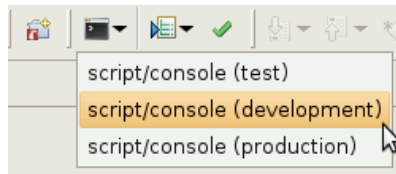
```
Restaurante.delete(1)
```

5.10 - Exercícios: Manipulando registros

Teste a manipulação de registros pelo console.

1) Insira um novo restaurante

a) Abra o console



b) Digite:

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

c) Olhe seu banco de dados:

```
mysql -u root  
  
use restaurante_development;  
select * from restaurantes;
```

d) Digite:

```
r.save
```

e) Olhe seu banco de dados novamente:

```
mysql -u root  
  
use restaurante_development;  
select * from restaurantes;
```

2) Atualize seu restaurante

a) Digite:

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

b) Olhe seu banco de dados novamente:

```
mysql -u root  
  
use restaurante_development;  
select * from restaurantes;
```

3) Vamos remover o restaurante criado:

a) Digite

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

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

```
mysql -u root  
  
use restaurante_development;  
select * from restaurantes;
```

5.11 - Exercícios Opcionais

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

5.12 - Finders

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

```
Restaurante.find(:all)    # retorna todos os registros
Restaurante.find(:first)  # retorna o primeiro registro
Restaurante.find(:last)   # retorna o último registro
```

Existem também atalhos para estes tipos de find:

```
Restaurante.all
Restaurante.first
Restaurante.last
```

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:

```
:conditions => "bla = 1 and xpto = 3"
:conditions => ["bla = ? and xpto = ?", 1, 3]
:conditions => ["bla = :b and xpto = :c", {:b => 1, :c => 3}]
:conditions => { :bla => 1, :xpto => 3 }
```

Essas quatro formas fazem a mesma coisa. Procuram por registros com o campo bla = 1 e o campo xpto = 3.

Existem ainda os chamados dynamic finders:

```
Restaurante.find(:all,
  :conditions => ["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, o método find aceita ainda outras opções:

- :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`.

Para Sabe 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.

5.13 - Exercícios: Buscas dinâmicas

1) Vamos testar os métodos de busca:

a) Abra o console (`ruby script/console`)

b) Digite:

```
Restaurante.find(:first)
```

c) Aperte **enter**

d) Digite:

```
Restaurante.find(:all)
```

e) Aperte **enter**

f) Digite:

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

g) Aperte **enter**

h) Digite:

```
Restaurante.find(:all, :conditions => ["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.find(:all, :order => "especialidade DESC", :limit => 1)
```

m) Aperte **enter**

5.14 - 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
```

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.

5.15 - 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"
```

- 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 do Aptana
- b) Digite:

```
r = Restaurante.new :nome => "fasano",
                   :endereco => "Av. dos Restaurantes, 126"

r.save
```

- c) Verifique a lista de erros, digitando:

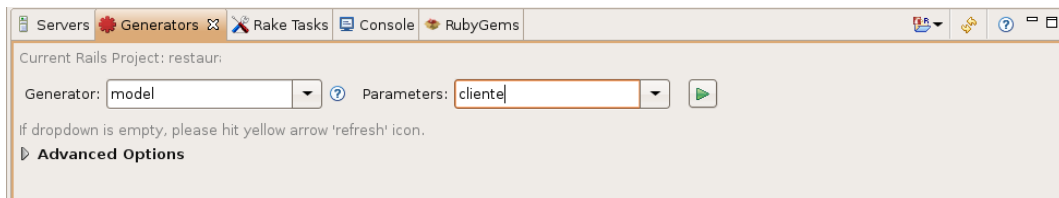
```
r.valid? # verifica se objeto passa nas validações
r.errors.empty? # retorna true/false indicando se há erros ou não
r.errors.count # retorna o número de erros
r.errors.on "nome" # retorna apenas o erro do atributo nome

r.errors.each {|field, msg| puts "#{field} - #{msg}" }
```

5.16 - Exercícios - Completando nosso modelo

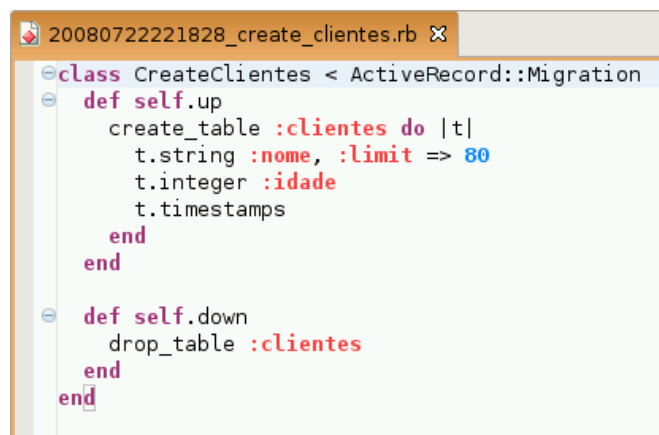
- 1) Vamos criar o nosso modelo Cliente, bem como sua migration:

- a) Entre na view “**Generator**”
- b) Escolha a opção “**model**”
- c) Digite o nome “**cliente**”
- d) Clique em “**go**”



- e) Abra o arquivo “**db/migrate/<timestamp>_create_clientes.rb**”
- f) Adicione as linhas:

```
create_table :clientes do |t|
  t.string :nome, :limit => 80
  t.integer :idade
  t.timestamps
end
```



- g) Abra a view “**Rake Tasks**”
- h) Escolha a opção “**db:migrate**”
- i) Aperte “**go**”
- j) Olhe no console o que foi feito

```
rake db:migrate
== 20080722221828 CreateClientes: migrating =====
-- create_table(:clientes)
   -> 0.1168s
== 20080722221828 CreateClientes: migrated (0.1174s) ==
```

- k) Olhe no banco de dados!

```
mysql> show tables;
+-----+
| Tables_in_restaurante_development |
+-----+
| clientes                           |
| restaurantes                       |
| schema_migrations                 |
+-----+
3 rows in set (0.00 sec)
```

2) Vamos agora fazer as validações no modelo “cliente”:

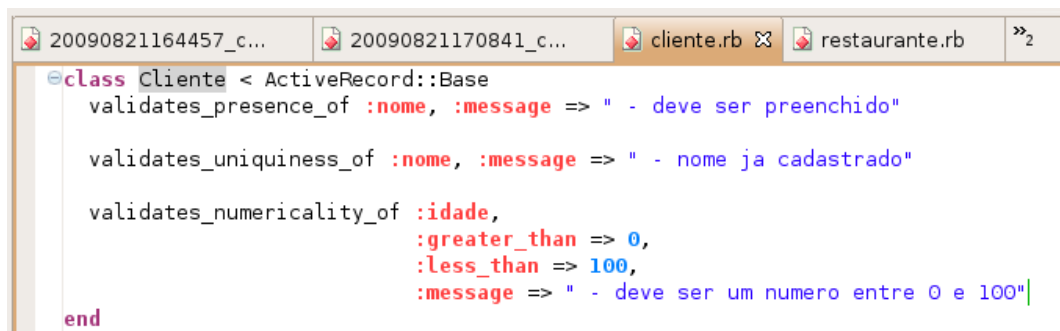
a) Abra o arquivo “**app/models/cliente.rb**”

b) Adicione as seguintes linhas:

```
validates_presence_of :nome, :message => " - deve ser preenchido"

validates_uniqueness_of :nome, :message => " - name já cadastrado"

validates_numericality_of :idade,
                          :greater_than => 0,
                          :less_than => 100,
                          :message => " - deve ser um número entre 0 e 100"
```



```
class Cliente < ActiveRecord::Base
  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 numero entre 0 e 100"
end
```

3) Vamos criar o modelo Prato, bem como sua migration:

a) Entre na view “**Generator**”

b) Escolha a opção “**model**”

c) Digite o nome “**prato**”

d) Clique em “**go**”

e) Abra o arquivo “**db/migrate/<timestamp>_create_pratos.rb**”

f) Adicione as linhas:

```
t.string :nome, :limit => 80
```

g) Abra a view “**Rake Tasks**”

h) Escolha a opção “**db:migrate**”

i) Aperte “**go**”

- j) Olhe no console o que foi feito
 - k) Olhe no banco de dados!
- 4) Vamos agora fazer as validações no modelo “prato”:

- a) Abra o arquivo “**app/models/prato.rb**”
- b) Adicione as seguintes linhas:

```
validates_presence_of :nome, :message => " - deve ser preenchido"

validates_uniqueness_of :nome, :message => " - nome já cadastrado"
```

- 5) Vamos criar o modelo Receita, bem como sua migration:

- a) Entre na view “**Generator**”
- b) Escolha a opção “**model**”
- c) Digite o nome “**receita**”
- d) Clique em “**go**”
- e) Abra o arquivo “**db/migrate/<timestamp>_create_receitas.rb**”
- f) Adicione as linhas:

```
t.text :conteudo
```

- g) Abra a view “**Rake**”
 - h) Escolha a opção “**db:migrate**”
 - i) Aperte “**go**”
 - j) Olhe no console o que foi feito
 - k) Olhe no banco de dados!
- 6) 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"
```

5.17 - O Modelo Qualificação

Antes de criarmos nosso modelo de Qualificação, repare que o Rails pluraliza os nomes de nossos modelos de forma automática. Por exemplo, o nome da tabela do modelo Cliente ficou clientes. No entanto, qualificacao é uma palavra que tem sua pluralização irregular (não basta adicionar um ‘s’ no final da palavra), e o Rails deve gerar a seguinte palavra pluralizada: ‘qualificacaos’, uma vez que estamos utilizando o português e ele possui regras de pluralização apenas para o inglês.

Para corrigir isso, vamos ter de editar o arquivo “**config/initializers/inflections.rb**” e inserir manualmente o plural da palavra ‘qualificacao’. Repare que nesse mesmo arquivo temos diversos exemplos comentados de como podemos fazer isso.

Quando inserirmos as linhas abaixo no final do arquivo, o Rails passará a utilizar esse padrão para a pluralização:

```
ActiveSupport::Inflector.inflections do |inflect|  
  inflect.irregular 'qualificacao', 'qualificacoes'  
end
```

5.18 - Exercícios - Criando o Modelo de Qualificação

1) Vamos corrigir a pluralização da palavra 'qualificacao'

a) Abra o arquivo “**config/initializers/inflections.rb**”

b) Adicione as seguintes linhas ao final do arquivo:

```
ActiveSupport::Inflector.inflections do |inflect|  
  inflect.irregular 'qualificacao', 'qualificacoes'  
end  
  
# Be sure to restart your server when you modify this file.  
  
# Add new inflection rules using the following format  
# (all these examples are active by default):  
# ActiveSupport::Inflector.inflections do |inflect|  
#   inflect.plural /^(ox)$/i, '\1en'  
#   inflect.singular /^(ox)en/i, '\1'  
#   inflect.irregular 'person', 'people'  
#   inflect.uncountable %w( fish sheep )  
# end  
ActiveSupport::Inflector.inflections do |inflect|  
  inflect.irregular 'qualificacao', 'qualificacoes'  
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.

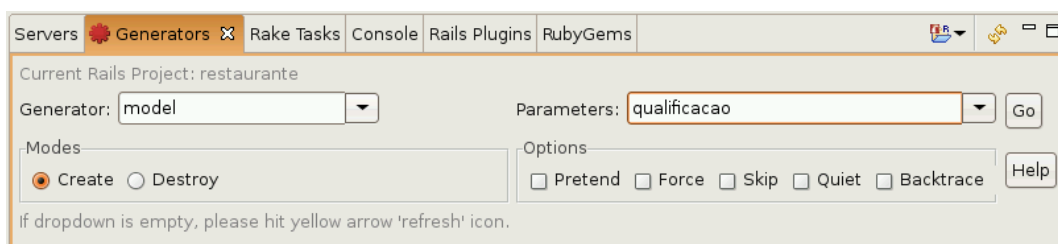
2) Vamos continuar com a criação do nosso modelo Qualificacao e sua migration.

a) Entre na view “**Generator**”

b) Escolha a opção “**model**”

c) Digite o nome “**qualificacao**”

d) Clique em “**go**”



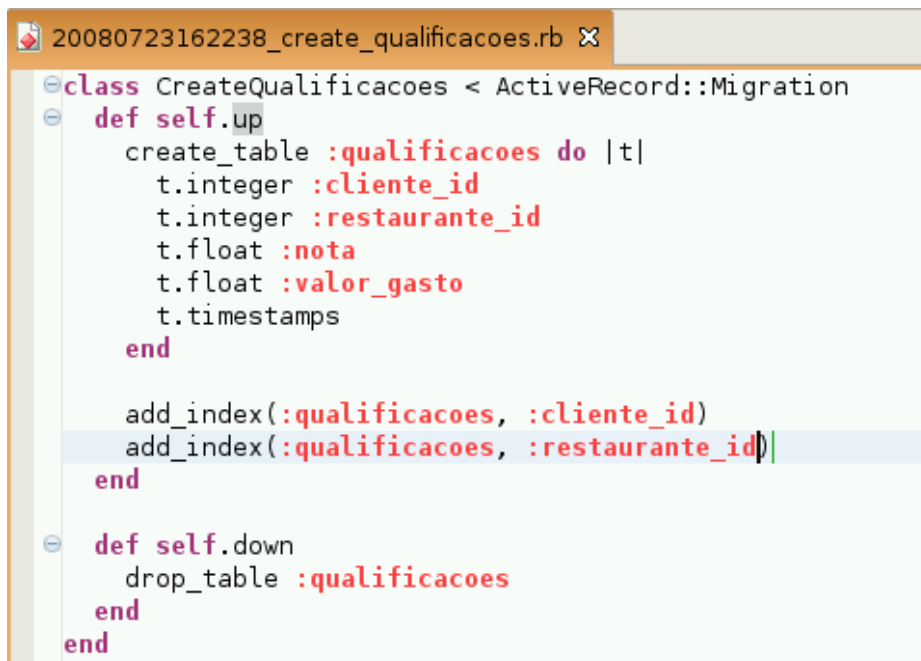
e) Abra o arquivo “**db/migrate/<timestamp>_create_qualificacoes.rb**”

f) Adicione as linhas:

```
t.integer :cliente_id
t.integer :restaurante_id
t.float :nota
t.float :valor_gasto
```

g) Adicione ainda as seguintes linhas depois de “**create_table**”: CUIDADO! Essas linhas fazem parte do **create_table**, devem ficar dentro do método.

```
add_index(:qualificacoes, :cliente_id)
add_index(:qualificacoes, :restaurante_id)
```



```
20080723162238_create_qualificacoes.rb X
class CreateQualificacoes < ActiveRecord::Migration
  def self.up
    create_table :qualificacoes do |t|
      t.integer :cliente_id
      t.integer :restaurante_id
      t.float :nota
      t.float :valor_gasto
      t.timestamps
    end

    add_index(:qualificacoes, :cliente_id)
    add_index(:qualificacoes, :restaurante_id)
  end

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

h) Abra a view “Rake”

i) Escolha a opção “**db:migrate**”

j) Aperte “go”

k) Olhe no console o que foi feito

```
rake db:migrate
== 20080723162238 CreateQualificacoes: migrating =====
-- create_table(:qualificacoes)
-> 0.2327s
-- add_index(:qualificacoes, :cliente_id)
-> 0.1299s
-- add_index(:qualificacoes, :restaurante_id)
-> 0.1025s
== 20080723162238 CreateQualificacoes: migrated (0.4660s) =====
```

l) Olhe no banco de dados

```
mysql> show tables;
+-----+
| Tables_in_restaurante_development |
+-----+
| clientes                           |
| qualificacoes                     |
| restaurantes                       |
| schema_migrations                 |
+-----+
4 rows in set (0.00 sec)
```

3) Vamos agora fazer as validações no modelo “qualificacao”:

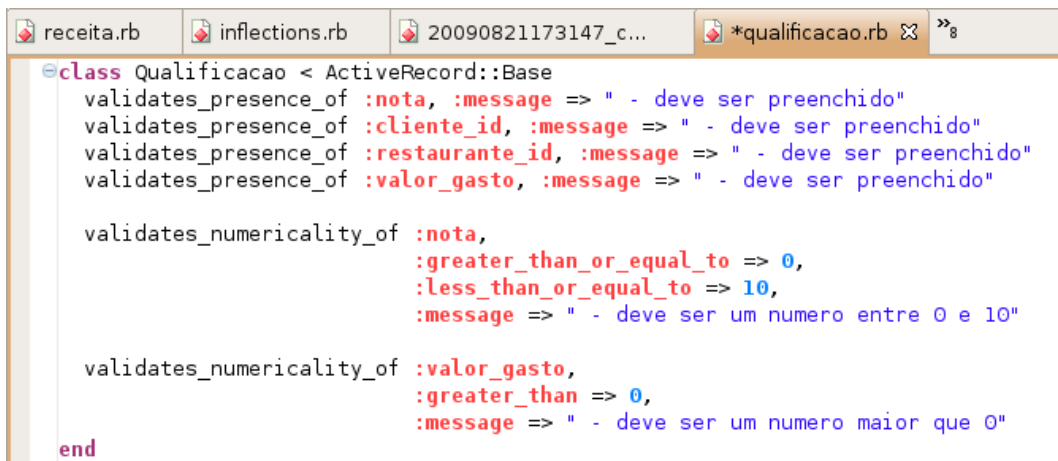
a) Abra o arquivo “app/models/qualificacao.rb”

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"
```



```
class Qualificacao < ActiveRecord::Base
  validates_presence_of :nota, :message => " - deve ser preenchido"
  validates_presence_of :cliente_id, :message => " - deve ser preenchido"
  validates_presence_of :restaurante_id, :message => " - deve ser preenchido"
  validates_presence_of :valor_gasto, :message => " - deve ser preenchido"

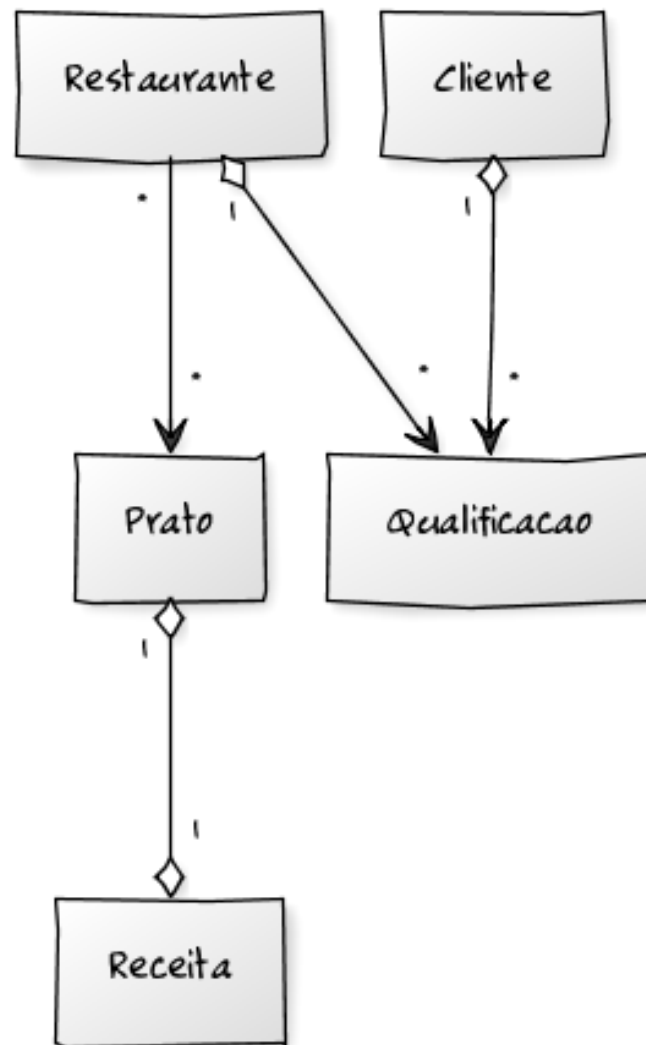
  validates_numericality_of :nota,
                            :greater_than_or_equal_to => 0,
                            :less_than_or_equal_to => 10,
                            :message => " - deve ser um numero entre 0 e 10"

  validates_numericality_of :valor_gasto,
                            :greater_than => 0,
                            :message => " - deve ser um numero maior que 0"
end
```

c) Abra o terminal do RadRails e digite:

```
"qualificacao".pluralize
```

5.19 - 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=

5.20 - 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 explícito. Exemplos:

```
restaurante.qualificações          # busca no banco de dados
restaurante.qualificações.size     # usa o cache
restaurante.qualificações.empty?   # usa o cache
restaurante.qualificações(true).size # força a busca no banco de dados
restaurante.qualificações          # usa o cache
```

5.21 - 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
```



```
*cliente.rb inflections.rb 20090821173147_c... *qualificacao.rb »
class Cliente < ActiveRecord::Base
  has_many :qualificacoes

  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 numero entre 0 e 100"
end
```

c) Abra o arquivo “**app/models/restaurante.rb**”

d) Adicione a seguinte linha:

```
has_many :qualificacoes
has_and_belongs_to_many :pratos
```

```
*cliente.rb *restaurante.rb 20090821173147_c... *qualificacao.rb »
class Restaurante < ActiveRecord::Base
  has_many :qualificacoes
  has_and_belongs_to_many :pratos

  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 => "endereco já cadastrado"

  validate :primeira_letra_deve_ser_maiuscula

  private
  def primeira_letra_deve_ser_maiuscula
    errors.add(:nome, "primeira letra deve ser maiuscula") unless nome =~ /[A-Z].*/
  end
end
```

e) Abra o arquivo “**app/models/qualificacao.rb**”

f) Adicione as seguintes linhas:

```
belongs_to :cliente
belongs_to :restaurante
```

```
cliente.rb  restaurante.rb  20090821173147_c...  qualificacao.rb  »
class Qualificacao < ActiveRecord::Base
  belongs_to :clientes
  belongs_to :restaurante

  validates_presence_of :nota, :message => " - deve ser preenchido"
  validates_presence_of :cliente_id, :message => " - deve ser preenchido"
  validates_presence_of :restaurante_id, :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 numero entre 0 e 10"

  validates_numericality_of :valor_gasto,
    :greater_than => 0,
    :message => " - deve ser um numero maior que 0"
end
```

g) Abra o arquivo **“app/models/prato.rb”**

h) Adicione as seguintes linhas:

```
has_and_belongs_to_many :restaurantes
has_one :receita
```

```
cliente.rb  restaurante.rb  receita.rb  qualificacao.rb  *prato.rb  »
class Prato < ActiveRecord::Base
  has_and_belongs_to_many :restaurantes
  has_one :receita

  validates_presence_of :nome, :message => " - deve ser preenchido"

  validates_uniqueness_of :nome, :message => " - nome ja cadastrado"
end
```

i) Abra o arquivo **“app/models/receita”**

j) Adicione a seguinte linha:

```
belongs_to: prato
```

```
cliente.rb  restaurante.rb  receita.rb  qualificacao.rb  *prato.rb  »
class Receita < ActiveRecord::Base
  belongs_to :prato

  validates_presence_of :conteudo, :message => " - deve ser preenchido"
end
```

2) Vamos criar a tabela para nosso relacionamento **has_and_belongs_to_many**:

- Entre na view **“Generators”**
- Escolha a opção **“migration”**
- Digite o nome **“createPratosRestaurantesJoinTable”**

- d) Clique em “go”
- e) Abra o arquivo “db/migrate/<timestamp>_create_pratos_restaurantes_join_table.rb”
- f) Adicione as linhas:

```
t.integer :prato_id  
t.integer :restaurante_id
```



```
class CreatePratosRestaurantesJoinTable < ActiveRecord::Migration  
  def self.up  
    create_table :pratos_restaurantes do |t|  
      t.integer :prato_id  
      t.integer :restaurante_id  
    end  
  end  
  
  def self.down  
    drop_table :pratos_restaurantes  
  end  
end
```

- g) Abra a view “rake”
 - h) Escolha a opção “db:migrate”
 - i) Aperte “go”
 - j) Olhe no console o que foi feito
 - k) 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.

5.22 - 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 Qualificacoes.all
  puts "restaurante  " + qualificacao.restaurante.nome
  puts "restaurante  " + 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:

```
Qualificacoes.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:

```
Qualificacoes.find(:all, :include => [ :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:

```
Qualificacoes.find(:all,
  :include => [ :cliente, { :restaurante => { :prato => :receita } } ])
```

5.23 - 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
  named_scope :massas, :conditions => { :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.find(:all, :conditions => ["nome like ?", '%x%'])
```

O método associado ao *named scope* criado retorna um objeto da classe *ActiveRecord::NamedScope*, 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
  named_scope :massas, :conditions => { :especialidade => 'massas' }
  named_scope :recentes, :conditions => [ "created_at > ?", 3.months.ago ]
  named_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
```

5.24 - 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
end
```

Controllers e Views

“A Inteligência é quase inútil para quem não tem mais nada”

– Carrel, Alexis

Nesse capítulo, você irá 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.

6.1 - 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.

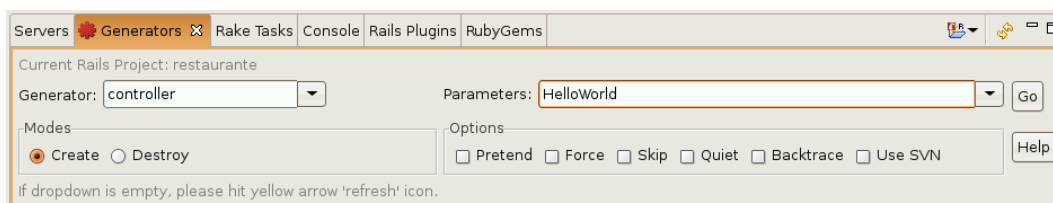
Controllers são classes que recebem uma ação de uma View e executam algum tipo de lógica ligada a um ou mais modelos. Em Rails esses controllers estendem a classe ApplicationController.

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).

6.2 - Hello World

Antes de tudo criaremos um controller que mostrará um “*Hello World*” para entender melhor como funciona essa idéia do mapeamento de urls.

Entre na view generators, escolha a opção controller e digite o nome “*HelloWorld*”.

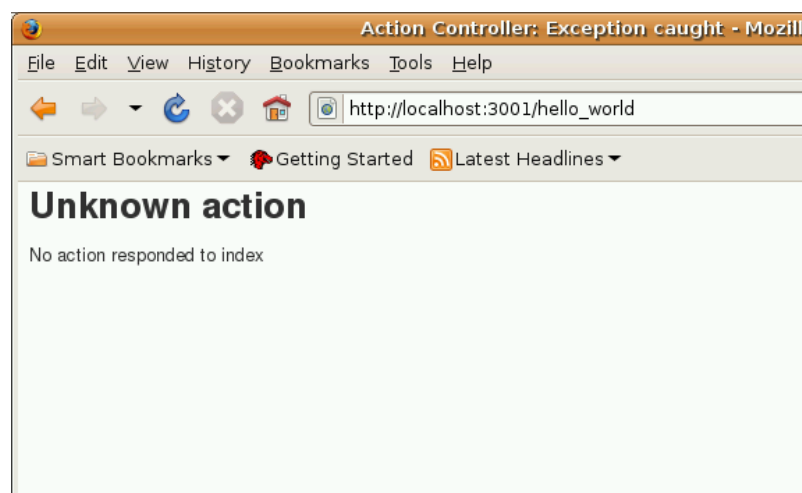


Veja que o Rails não gera apenas o Controller, mas também outros arquivos.

```
>script/generate controller HelloWorld
exists app/controllers/
exists app/helpers/
create app/views/hello_world
exists test/functional/
create app/controllers/hello_world_controller.rb
create test/functional/hello_world_controller_test.rb
create app/helpers/hello_world_helper.rb
```

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.



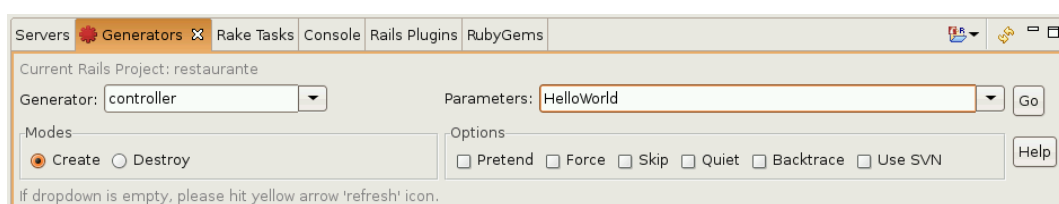
Além de não dizer qual a action na URL, não escrevemos nenhuma action no controller.

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.

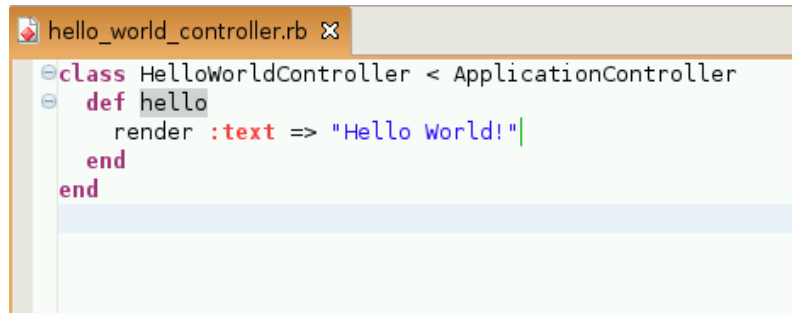
6.3 - Exercícios: Criando o controlador

- 1) Crie um controller que mostre na tela a mensagem “Hello World”
 - a) Entre na view generators, e escolha a opção “controller”
 - b) Digite o nome do controller como “HelloWorld”
 - c) Clique em “go”.

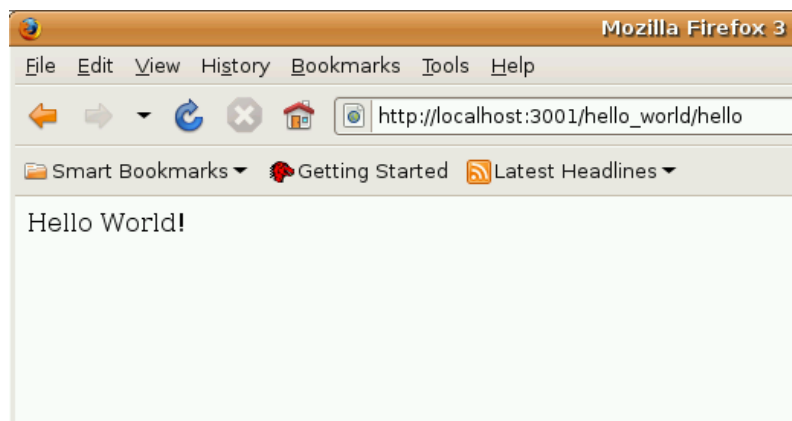


- d) Entre no seu novo controller (**app/controllers/hello_world_controller.rb**)
e) Inclua o método "hello":

```
def hello
  render :text => "Hello World!"
end
```



- f) Confira o link http://localhost:3000/hello_world/hello.



6.4 - Redirecionamento de Action e Action padrão

Mas não há necessidade de toda vez escolher qual action será executada. Temos a opção de escrever uma action padrão que será invocada sempre que nenhuma outra for definida na url sendo acessada.

Para definir uma action padrão temos que dar o nome do método de `index`:

```
def index
  render :text => "Action Padrão!"
end
```

Dentro de uma action, podemos redirecionar a saída para uma outra action. No exemplo do Hello World, ao chamar o comando `redirect_to` e passar como argumento a action desejada (`:action => "hello"`

), o Rails fará o redirecionamento no cliente.

```
def index
  redirect_to(:action => "hello")
end
```

Ao acessar http://localhost:3000/hello_world, o servidor nos redireciona para a página http://localhost:3000/hello_world/hello, mostrando um "Hello World" igual ao anterior.



Ao invés de fazer o redirecionamento, podemos chamar outra action diretamente. Assim não vamos ser redirecionados e a URL continuará a mesma.

```
def index
  hello
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.

6.5 - 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 idéia é que o HTML serve como um template, e outros elementos são dinamicamente inseridos em tempo de renderização.

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. Existe ainda o “<%=h”, que traduz qualquer caracter estranho (“<”, “>”, “””, “'”, etc) para seu código HTML equivalente.

É importante notar, que todos os atributos de instância (@variavel) de um controlador estão disponíveis em sua view. Além disso, a view deve ter o mesmo 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**.

Podemos começar precisamos que uma view seja renderizada para pegar os valores de um novo restaurante. (Essa view será apresentada após o código da action new, que faremos mais a frente):

```
<form action='/restaurantes/create'>
  Nome: <input type='text' name='nome' />
  <input type='submit' value='Create' />
</form>
```

E agora, para receber este valor 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
```

O problema desta abordagem é que precisaríamos recuperar os valores enviados pelo formulário, um a um. Para um formulário com 15 campos, teríamos 15 linhas apenas para recuperar as informações submetidas!

Para simplificar esta tarefa, o Rails oferece a possibilidade de utilizar algo parecido com um hash para os valores dos atributos:

```
<form action='/restaurantes/create'>
  Nome: <input type='text' name='restaurante[nome]' />
  Endereço: <input type='text' name='restaurante[endereco]' />
  Especialidade: <input type='text' name='restaurante[especialidade]' />
  <input type='submit' value='Create' />
</form>
```

Desta forma, podemos receber todos os valores como um hash nos controladores:

```
class RestaurantesController < ApplicationController
  def create
    valores = params['restaurante']
  end
end
```

O mais interessante é que as classes ActiveRecord já aceitam um hash com os valores iniciais do objeto, tanto no método `new`, no método `create` (que já salva), quanto no método `update_attributes`:

```
Restaurante.create(params['restaurante'])
```

6.6 - 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:

- `list`: exibe todos os itens
- `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

Desejamos listar todos os restaurantes do nosso Banco de Dados, e portanto criaremos a action `list`. Como desejamos que o comportamento padrão do nosso controlador `restaurante` seja exibir a listagem de restaurantes, podemos renomeá-la para `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.find(:all, :order => "nome")
end
```

Para exibir um restaurante específico, precisamos saber qual restaurante buscar. Essa informação vem no “id” da url, e contem o id do restaurante que desejamos. Para acessá-la, podemos usar como parâmetro do método find `params[:id]`, que recupera as informações passadas no id da url.

Agora, podemos criar nossa action `show`:

```
def show
  @restaurante = Restaurante.find(params[:id])
end
```

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
```

Uma vez que o usuário do nosso sistema tenha preenchido as informações do novo restaurante e deseje salvá-las, enviará uma requisição à nossa action `create`, passando como parâmetro na requisição, o novo restaurante a ser criado. Vamos criar nossa action:

```
def create
  @restaurante = Restaurante.new(params[:restaurante])
  @restaurante.save
end
```

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
```

Uma vez que o usuário tenha atualizado as informações do restaurante e deseje salvá-las, enviará uma requisição à nossa action `update` passando no id da url o id do restaurante a ser editado, bem como o restaurante que será “colado” em seu lugar:

```
def update
  @restaurante = Restaurante.find(params[:id])
  @restaurante.update_attributes(params[:restaurante])
end
```

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
```

6.7 - Exercícios: Controlador do Restaurante

- 1) Gere um controller para o modelo restaurante:
 - a) Entre na view Generators;
 - b) Escolha a opção controller;
 - c) Digite "restaurantes";
 - d) Aperte "go".
- 2) Crie as actions CRUD para o controller criado:
 - a) Abra o seu controller de Restaurantes (**app/controllers/restaurantes_controllers.rb**)
 - b) Insira no controller a actions CRUD:

```
class RestaurantesController < ApplicationController
  def index
    @restaurantes = Restaurante.find(:all, :order => "nome")
  end

  def show
    @restaurante = Restaurante.find(params[:id])
  end

  def new
    @restaurante = Restaurante.new
  end

  def create
    @restaurante = Restaurante.new(params[:restaurante])
    if @restaurante.save
      redirect_to(:action => "show", :id => @restaurante)
    else
      render :action => "new"
    end
  end

  def edit
    @restaurante = Restaurante.find(params[:id])
  end

  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
end
```

```
def destroy
  @restaurante = Restaurante.find(params[:id])
  @restaurante.destroy

  redirect_to(:action => "index")
end
end
```

6.8 - 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.

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.

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 reescrever nossa view:

```
<% form_tag :action => 'create' do %>
  Nome: <%= text_field :restaurante, :nome %>
  Endereço: <%= text_field :restaurante, :endereco %>
  Especialidade: <%= text_field :restaurante, :especialidade %>
  <%= submit_tag 'Create' %>
<% 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, :url => { :action => 'create' } do |f| %>
  Nome: <%= f.text_field :nome %>
  Endereço: <%= f.text_field :endereco %>
  Especialidade: <%= f.text_field :especialidade %>
  <%= submit_tag 'Create' %>
<% 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.

Por último, poderíamos querer ver possíveis mensagens com problemas de validação. O módulo `ActionViewHelpers::ActiveRecordHelpers` contém o helper `error_messages_for`, que recebe o nome do modelo do qual serão exibidos todos os problemas de validação, caso existam.

```
<%= error_messages_for :restaurante %>
```

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
    Helper Method "algum conteudo dinamico"
  end
end
```

E em alguma das views deste controlador:

```
Helper Method<%Helper Method= teste %Helper Method>
```

6.9 - Exercícios: Utilizando helpers para criar as views

1) Vamos criar as views do restaurante:

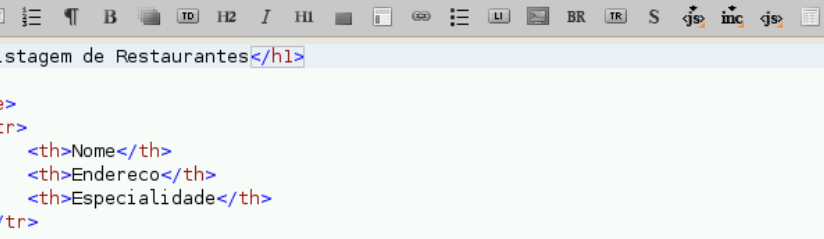
- Crie o arquivo **app/views/restaurantes/index.html.erb**. Você pode clicar com o botão direito do mouse em **app/views/restaurantes** e escolher a opção New => ERB/RHTML File, ou dentro do corpo da action index, pressionar "Ctrl+Shift+V". Em ambos os casos, escolha o nome **index.html.erb**
- Digite o conteúdo abaixo:

```
<h1>Listagem de Restaurantes</h1>

<table>
  <tr>
    <th>Nome</th>
    <th>Endereço</th>
    <th>Especialidade</th>
  </tr>

  <% for restaurante in @restaurantes %>
    <tr>
      <td><%=h restaurante.nome %></td>
      <td><%=h restaurante.endereco %></td>
      <td><%=h restaurante.especialidade %></td>
      <td><%= link_to 'Show', { :action => 'show', :id => restaurante } %></td>
      <td><%= link_to 'Edit', { :action => 'edit', :id => restaurante } %></td>
      <td><%= link_to 'Destroy', { :action => 'destroy', :id => restaurante } %></td>
    </tr>
  <% end %>
</table>

<br/>
<%= link_to 'New', { :action => 'new' } %>
```



```

<h1>Listagem de Restaurantes</h1>

<table>
  <tr>
    <th>Nome</th>
    <th>Endereco</th>
    <th>Especialidade</th>
  </tr>
  <% for restaurante in @restaurantes %>
    <tr>
      <td>=<h> restaurante.nome %></td>
      <td>=<h> restaurante.endereco %></td>
      <td>=<h> restaurante.especialidade %></td>
      <td>=<link_to 'Show', {:action => 'show', :id => restaurante} %></td>
      <td>=<link_to 'Edit', {:action => 'edit', :id => restaurante} %></td>
      <td>=<link_to 'Destroy', {:action => 'destroy', :id => restaurante} %></td>
    </tr>
  <% end %>
</table>

<br/>
<%= link_to 'New', {:action => 'new'} %>

```

- c) Teste agora entrando em: <http://localhost:3000/restaurantes> (talvez seja necessário reiniciar o servidor)



- d) Crie o arquivo **app/views/restaurantes/show.html.erb**
- e) Digite o conteúdo abaixo:

```
<h1>Exibindo Restaurante</h1>

<p>
  <b>Nome: </b>
  <%=h @restaurante.nome %>
</p>
```



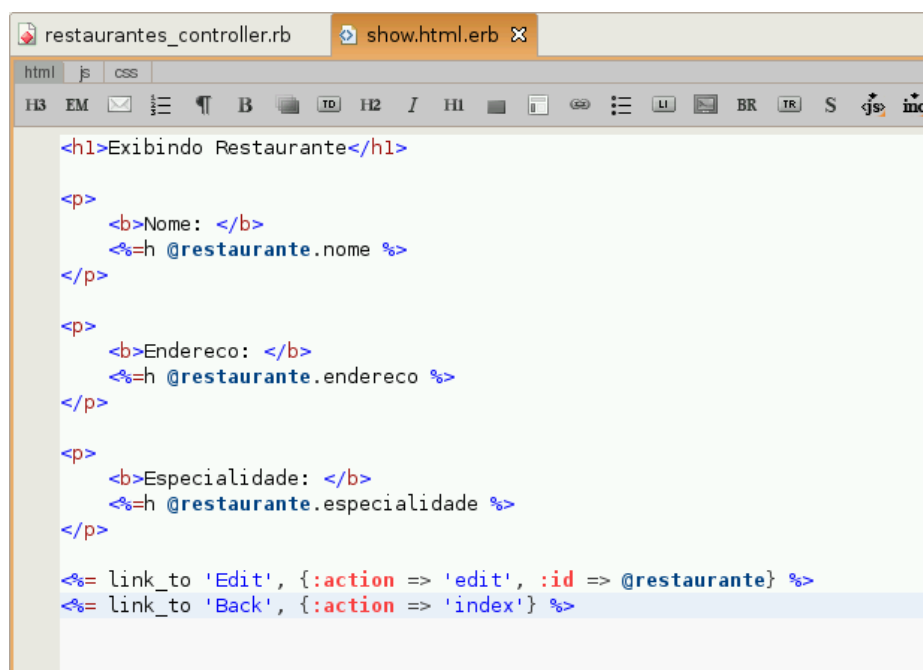
```

</p>
  <b>Endereço: </b>
  <%=h @restaurante.endereco %>
</p>

<p>
  <b>Especialidade: </b>
  <%=h @restaurante.especialidade %>
</p>

<%= link_to 'Edit', { :action => 'edit', :id => @restaurante } %>
<%= link_to 'Back', { :action => 'index' } %>

```



f) Crie o arquivo **app/views/restaurantes/new.html.erb**

g) Digite o conteúdo abaixo:

```

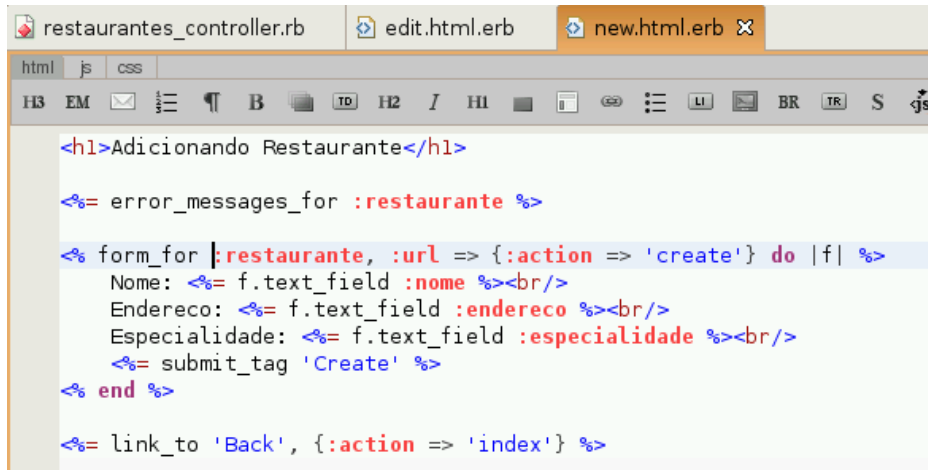
<h1>Adicionando Restaurante</h1>

<%= error_messages_for :restaurante %>

<% form_for :restaurante, :url => { :action => 'create' } do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= submit_tag 'Create' %>
<% end %>

<%= link_to 'Back', { :action => 'index' } %>

```



```

restaurantes_controller.rb  edit.html.erb  new.html.erb
html  js  css
H3 EM  B  TD H2 I H1  BR TR S  js
<h1>Adicionando Restaurante</h1>

<%= error_messages_for :restaurante %>

<% form_for |:restaurante, :url => {:action => 'create'} do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereco: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= submit_tag 'Create' %>
<% end %>

<%= link_to 'Back', {:action => 'index'} %>
  
```

h) Crie o arquivo **app/views/restaurantes/edit.html.erb**

i) Digite o conteúdo abaixo:

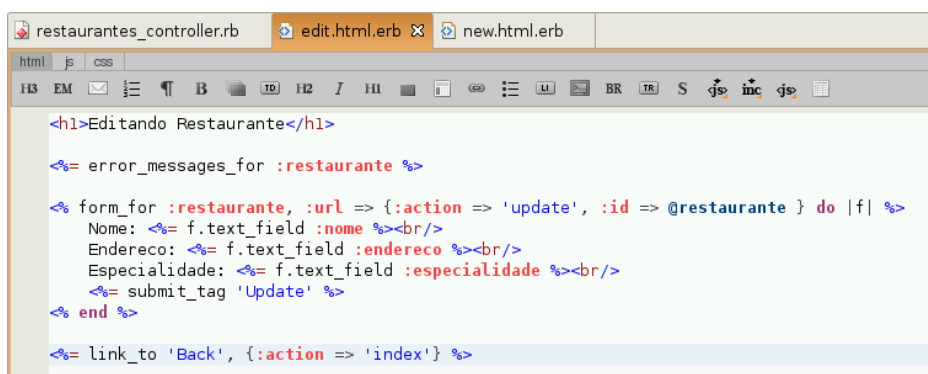
```

<h1>Editando Restaurante</h1>

<%= error_messages_for :restaurante %>

<% form_for :restaurante, :url => { :action => 'update', :id => @restaurante } do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= submit_tag 'Update' %>
<% end %>

<%= link_to 'Back', { :action => 'index' } %>
  
```



```

restaurantes_controller.rb  edit.html.erb  new.html.erb
html  js  css
H3 EM  B  TD H2 I H1  BR TR S  js  inc  js
<h1>Editando Restaurante</h1>

<%= error_messages_for :restaurante %>

<% form_for :restaurante, :url => {:action => 'update', :id => @restaurante } do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= submit_tag 'Update' %>
<% end %>

<%= link_to 'Back', { :action => 'index' } %>
  
```

j) Teste suas views: <http://localhost:3000/restaurantes> Não esqueça que existe uma validação para primeira letra maiúscula no nome do restaurante.

6.10 - 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:

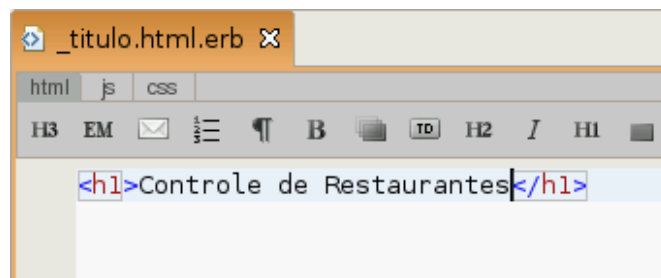
```
render :partial => "meupartial"
```

6.11 - Exercícios: Customizando o cabeçalho

1) Vamos criar um partial:

- Crie o arquivo: **app/views/restaurantes/_titulo.html.erb**
- Coloque o seguinte conteúdo:

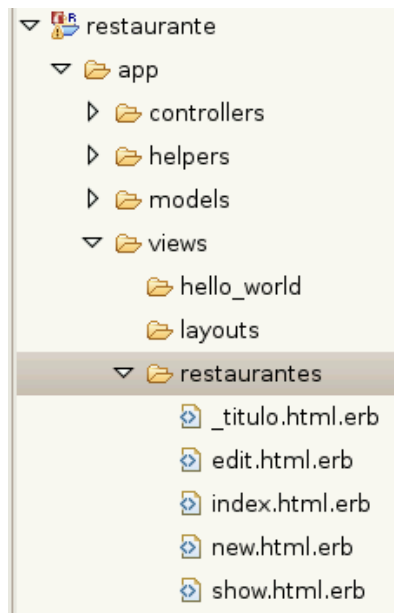
```
<h1>Controle de Restaurantes</h1><br>
```



- Abra todos os arquivos criados no exercício anterior (**app/views/restaurantes/***)
- Insira a seguinte linha no início:

```
<%= render :partial => "titulo" %>
```

Seu **app/views/restaurantes** deve estar com os seguintes arquivos:



e) Teste suas views: <http://localhost:3000/restaurantes>

6.12 - Layout

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>
    <p style="color: green"><%= flash[:notice] %></p>

    <%= yield %>
  </body>
</html>
```

Tudo o que está nesse arquivo pode ser modificado, com exceção do `<%= yield %>`, que renderiza cada view do nosso controlador. Também deixaremos a linha `<p style="color: green"><%= flash[:notice] %></p>`, pois ela é a responsável por exibir as mensagens do escopo flash.

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.

6.13 - Exercícios: Criando o header

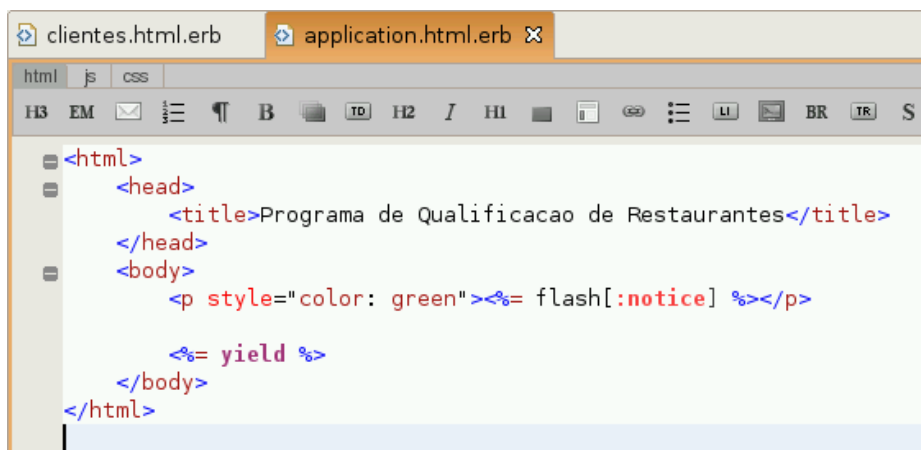
1) Nesse exercício, vamos utilizar o layout application para alterar o título de todas as janelas de nossa aplicação:

a) Crie o arquivo **app/views/layouts/application.html.erb**

b) Insira o seguinte código:

```
<html>
  <head>
    <title>Programa de Qualificação de Restaurantes</title>
  </head>
  <body>
    <p style="color: green"><%= flash[:notice] %></p>

    <%= yield %>
  </body>
</html>
```



c) Teste suas views: <http://localhost:3000/restaurantes>. Repare no título das janelas.

6.14 - Outras formas de gerar a View

O Rails já vem com suporte a outros *handlers* para geração de views. Além do ERB, podemos também usar o **Builder** e o **RJS**, que veremos mais adiante.

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>
```

6.15 - 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 Logger= :debug  
  config.log_path Logger= Logger.logLogger/debug.logLogger  
  # ...  
end
```

Mais detalhes sobre a customização do log podem ser encontrados no wiki oficial do Rails:
<http://wiki.rubyonrails.org/rails/show/HowtoConfigureLogging>

Rotas

“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 `ActionController::Routing` e as rotas podem ser customizadas no arquivo `config/routes.rb`.

7.1 - routes.rb

Por padrão, o arquivo *routes.rb* vem como a rota padrão registrada:

```
ActionController::Routing::Routes.draw do |map|
  map.connect 'inicio', :controller => 'restaurantes', :action => 'index'
end
```

`map.connect` cria uma nova rota, 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:

```
map.connect 'categorias/:nome', :controller => 'categorias', :action => 'show'
```

Neste caso, o parâmetro de requisição `params['nome']` é extraído da própria url!

Rotas padrão

Você pode ter notado no seu `config/routes.rb`, que o Rails instala duas rotas por padrão:

```
map.connect Rotas padrão':controllerRotas padrão/:actionRotas padrão/:idRotas padrão'  
map.connect Rotas padrão':controllerRotas padrão/:actionRotas padrão/:id.:formatRotas  
padrão'
```

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.

7.2 - 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:

```
map.connect 'blog/:ano', :controller => 'posts', :action => 'list'
```

Ou ainda para um mês específico:

```
map.connect 'blog/:ano/:mes', :controller => 'posts', :action => 'list'
```

Os parâmetros capturados pela url podem ter ainda valores *default*:

```
map.connect 'blog/:ano', :controller => 'posts', :action => 'list', :ano => 2008
```

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 2008.

7.3 - Named Routes

Cada uma das rotas pode ter um nome único:

```
map.posts 'blog/:ano', :controller => 'posts', :action => 'list'
```

O funcionamento é o mesmo de antes, com a diferença de que ao invés de usarmos `map.connect`, 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
```

7.4 - REST - map.resource

REST é um modelo arquitetural para sistemas distribuídos. A idéia 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 idéias 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  
map.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.

```
script/generate resource comentario
```

O gerador de scaffolds do Rails 2.0 em diante, também usa o modelo REST:

```
script/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
```

7.5 - 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:

```
map.resources :comentarios, :member => { :desabilita => :post }  
# url: /comentarios/:id/desabilita, named_route: desabilita_comentario_path
```

:member define actions que atuam sobre um recurso específico: `/comentarios/1/desabilita`. Recebe um hash na forma `:action => :method`, onde `:method` pode ser `:get`, `:post`, `:put`, `:delete` ou `:any`.

A opção `:collection`, serve para definir actions extras que atuem sobre o conjunto inteiro de resources. Definirá rotas do tipo `/comentarios/action`.

```
map.resources :comentarios, :collection => { :feed => :get }  
# url: /comentarios/feed, named_route: feed_comentarios_path
```

Para todas as actions extras, são criadas *Named Routes* adequadas. Use o rake `routes` como referência para conferir os nomes dados as rotas criadas.

7.6 - Diversas Representações

Um controlador pode ter diversos resultados. Em outras palavras, controladores podem responder de diversas maneiras, através do método `respond_to`:

```
class MeuController < ApplicationController  
  def list  
    respond_to do |format|  
      format.html  
      format.js do  
        render :update do |page|  
          page.insert_html :top, 'div3', "Novo conteudo"  
        end  
      end  
      format.xml  
    end  
  end  
end
```

O convenção para o nome da view de resultado é sempre:

`app/views/:controller/:action.:format.:handler`

Os *handlers* disponíveis por padrão no Rails são: **erb**, **builder** e **rjs**. Os formatos instalados por padrão ficam na constante `Mime::SET` e você pode instalar outros pelo arquivo `config/initializers/mime_types.rb`.

7.7 - 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
map.resources :restaurantes do |restaurante|
  restaurante.resources :qualificacoes
end
```

A rota acima automaticamente cria as rotas para `qualificacoes` 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']`).

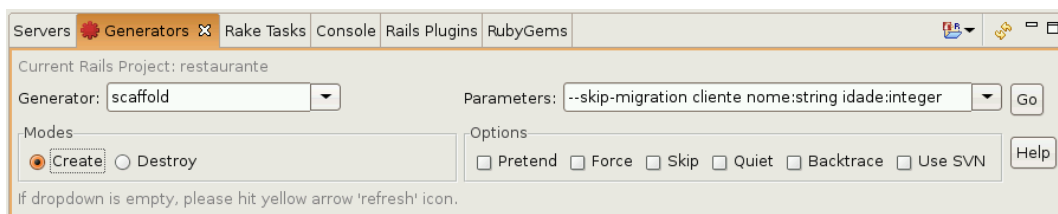
Completando o Sistema

“O êxito parece doce a quem não o alcança”
– Dickinson, Emily

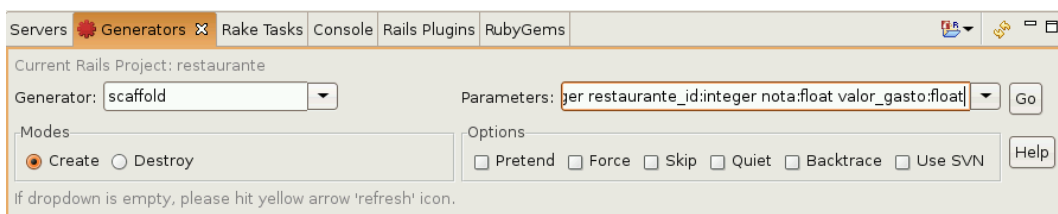
8.1 - Exercícios

1) Vamos gerar os outros controllers e views usando o scaffold:

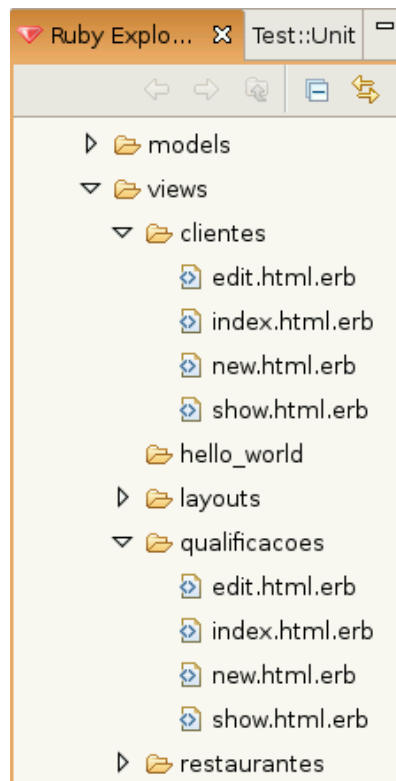
- Abra a view “Generators”
- Escolha a opção “scaffold”
- Primeiro vamos gerar para **cliente**. Passe como parâmetro: `--skip-migration cliente nome:string idade:integer`
- Clique em “go”



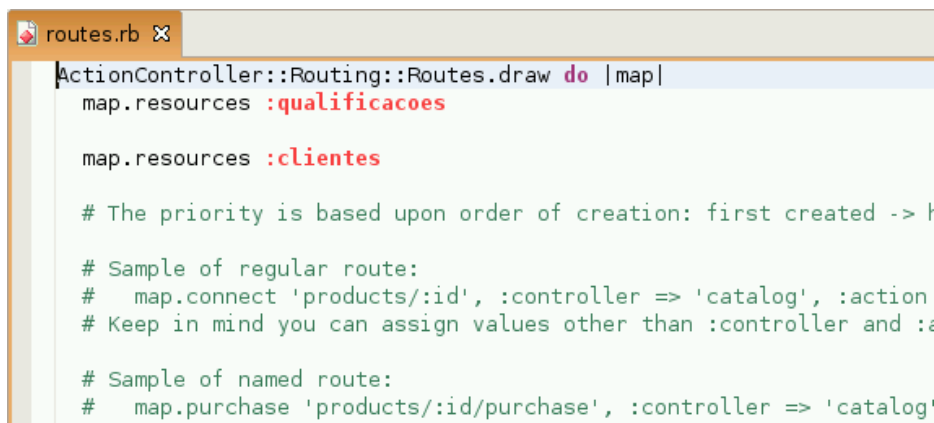
- Escolha a opção scaffold novamente
- Agora vamos gerar para *****qualificacao***. Passe como parâmetro: `--skip-migration qualificacao cliente_id:integer restaurante_id:integer nota:float valor_gasto:float`
- Clique em “go”



- Olhe as views criadas (**app/views/clientes** e **app/views/qualificacoes**)



i) Olhe as rotas criadas (**app/config/routes.rb**)



j) Apague os layouts criados pelo scaffold. Queremos utilizar nosso *application layout* criado anteriormente (**app/views/layouts/clientes.html.erb** e **app/views/layouts/qualificacoes.html.erb**)

k) 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

l) Reinicie o servidor

m) Teste: **http://localhost:3000/clientes** e **http://localhost:3000/qualificacoes**

Note que precisamos utilizar a opção “`--skip-migration`” 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>`:

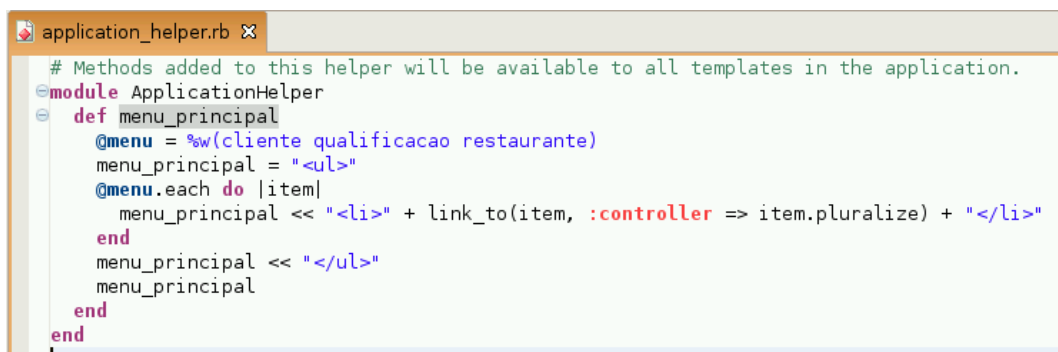
```
css scaffold<%css scaffold= stylesheet_link_tag css scaffold'scaffoldcss
scaffold' %css scaffold>
```

2) Vamos agora incluir **Helpers** no nosso **Layout**, para poder navegar entre Restaurante, Cliente e Qualificação sem precisarmos digitar a url:

a) Abra o Helper “applications”: “**app/helpers/application_helper.rb**”

b) Digite o seguinte método:

```
def menu_principal
  @menu = %w(cliente qualificacao restaurante)
  menu_principal = "<ul>"
  @menu.each do |item|
    menu_principal << "<li>" + link_to(item, :controller => item.pluralize) + "</li>"
  end
  menu_principal << "</ul>"
  menu_principal
end
```

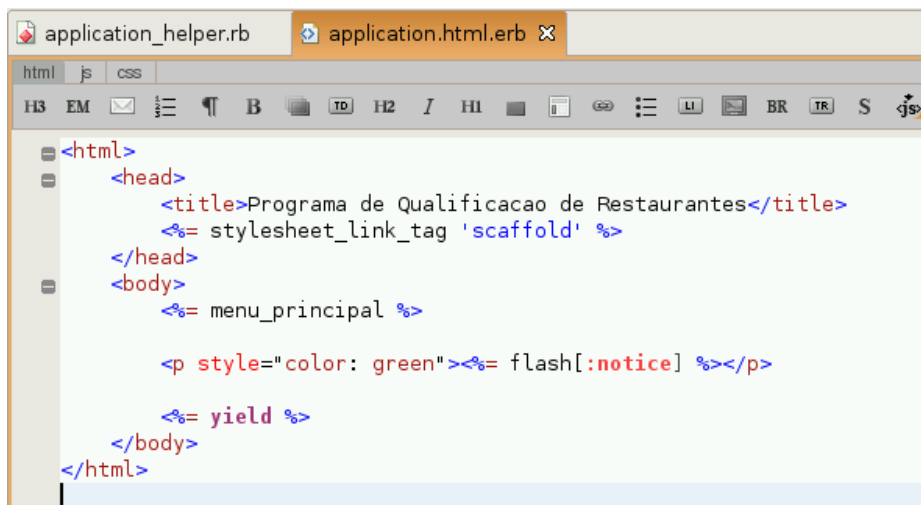


c) Adicione a chamada ao método `<%=menu_principal %>` no nosso layout (**app/views/layouts/application.html.erb**), ficando desse jeito:

```
...
<body>
  <%= menu_principal %>

  <p style="color: green"><%= flash[:notice] %></p>

  <%= yield %>
</body>
...
```

```
application_helper.rb application.html.erb
html js css
H3 EM [icon] B TD H2 I H1 [icon] [icon] BR TR S [icon]
<html>
<head>
<title>Programa de Qualificacao de Restaurantes</title>
<%= stylesheet_link_tag 'scaffold' %>
</head>
<body>
<%= menu_principal %>

<p style="color: green"><%= flash[:notice] %></p>

<%= yield %>
</body>
</html>
```

d) Teste: <http://localhost:3000/restaurantes>

8.2 - 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.find(:all, :order => :nome).collect
  {|p| [p.nome, p.id]}) %>
```

em substituição ao:

```
<%= f.text_field :cliente_id %>
```

Mas existe um outro **FormHelper** mais elegante, que produz o mesmo efeito, o `collection_select`:

```
<%= collection_select(:qualificacao, :cliente_id,
  Cliente.find(:all, :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 ActiveRecord, e com isso fazer apenas:

```
<%= f.collection_select(:cliente_id,
  Cliente.find(:all, :order => :nome),
  :id, :nome, {:prompt => true}) %>
```

8.3 - Exercícios

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/new.html.erb**

b) Troque a linha:

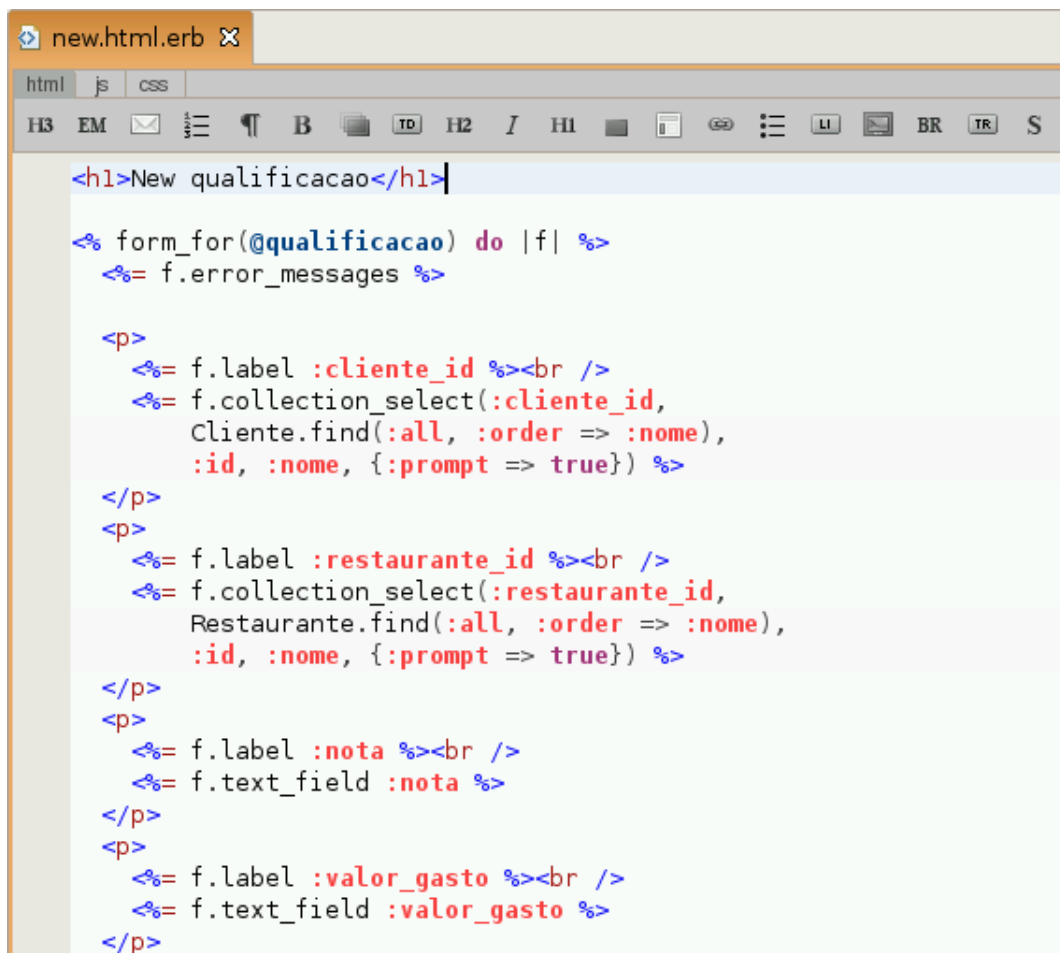
```
<%= f.text_field :cliente_id %>
por:

<%= f.collection_select(:cliente_id,
  Cliente.find(:all, :order => :nome),
  :id, :nome, {:prompt => true}) %>
```

c) Troque a linha:

```
<%= f.text_field :restaurante_id %>
por:

<%= f.collection_select(:restaurante_id,
  Restaurante.find(:all, :order => :nome),
  :id, :nome, {:prompt => true}) %>
```



```
<h1>New qualifiacao</h1>

<%= form_for(@qualifiacao) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :cliente_id %><br />
    <%= f.collection_select(:cliente_id,
      Cliente.find(:all, :order => :nome),
      :id, :nome, {:prompt => true}) %>
  </p>
  <p>
    <%= f.label :restaurante_id %><br />
    <%= f.collection_select(:restaurante_id,
      Restaurante.find(:all, :order => :nome),
      :id, :nome, {:prompt => true}) %>
  </p>
  <p>
    <%= f.label :nota %><br />
    <%= f.text_field :nota %>
  </p>
  <p>
    <%= f.label :valor_gasto %><br />
    <%= f.text_field :valor_gasto %>
  </p>
```

d) Faça o mesmo com o arquivo **app/views/qualificacoes/edit.html.erb**

e) Teste: <http://localhost:3000/qualificacoes>

2) 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:

```
<%=h @qualificacao.cliente_id %>
```

por:

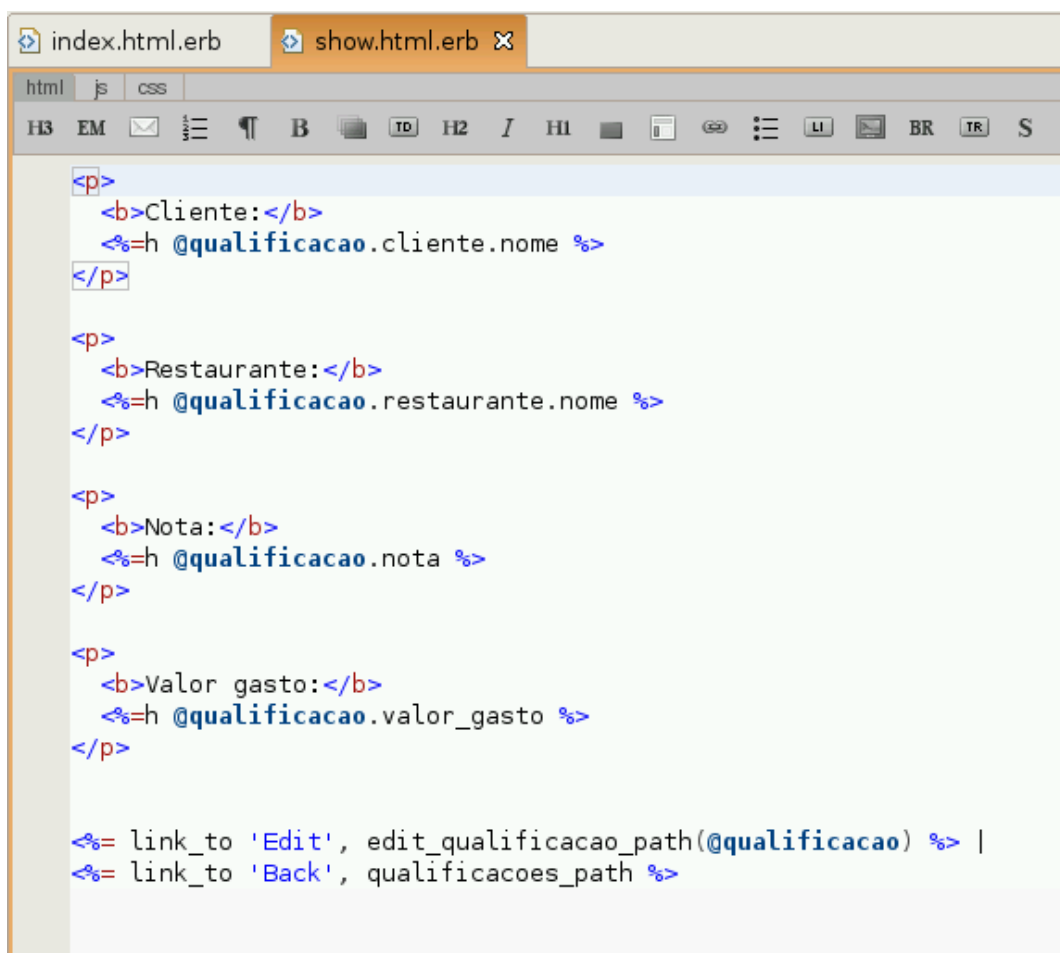
```
<%=h @qualificacao.cliente.nome %>
```

c) Troque a linha:

```
<%=h @qualificacao.restaurante_id %>
```

por:

```
<%=h @qualificacao.restaurante.nome %>
```



```
index.html.erb show.html.erb X
html js css
H3 EM [icon] B TD H2 I H1 [icon] [icon] LI [icon] BR TR S
<p>
  <b>Cliente:</b>
  <%=h @qualificacao.cliente.nome %>
</p>

<p>
  <b>Restaurante:</b>
  <%=h @qualificacao.restaurante.nome %>
</p>

<p>
  <b>Nota:</b>
  <%=h @qualificacao.nota %>
</p>

<p>
  <b>Valor gasto:</b>
  <%=h @qualificacao.valor_gasto %>
</p>

<%= link_to 'Edit', edit_qualificacao_path(@qualificacao) %> |
<%= link_to 'Back', qualificacoes_path %>
```

d) Abra o arquivo **app/views/qualificacoes/index.html.erb**

e) Troque as linhas:

```
<td><%=h @qualificacao.cliente_id %></td>
```

```
<td><%=h @qualificacao.restaurante_id %></td>
```

por:

```
<td><%=h @qualificacao.cliente.nome %></td>
```

```
<td><%=h @qualificacao.restaurante.nome %></td>
```

```
index.html.erb show.html.erb
html js css
HB EM 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 10
```

f) Teste: <http://localhost:3000/qualificacoes>

3) 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/new.html.erb**

b) Troque as linhas

```
<p>
  <%= f.label :cliente_id %><br />
  <%= f.collection_select(:cliente_id,
    Cliente.find(:all, :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.find(:all, :order => :nome),
    :id, :nome, {:prompt => true}) %></p>
<% end %>
```

c) Troque as linhas

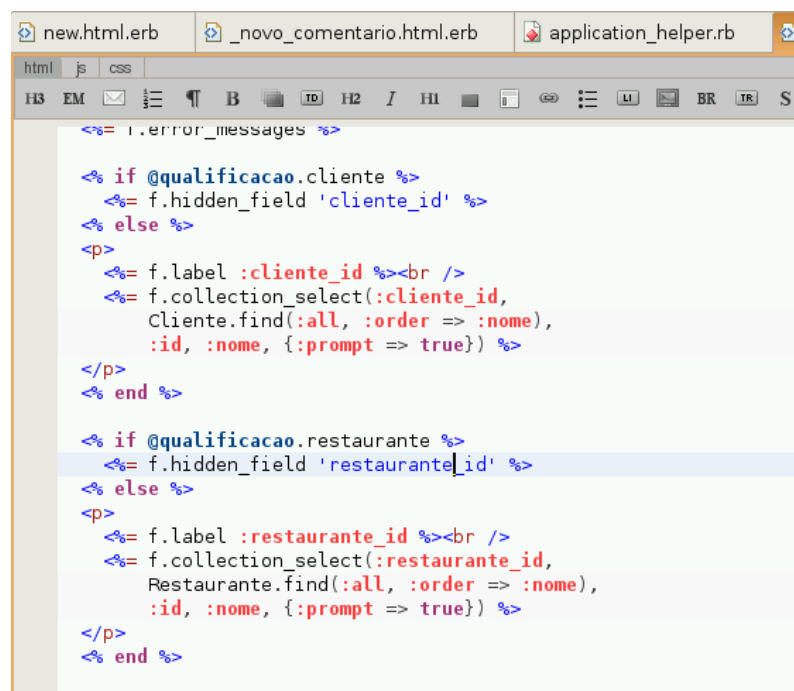
```
<p>
  <%= f.label :restaurante_id %><br />
  <%= f.collection_select(:restaurante_id,
    Restaurante.find(:all, :order => :nome),
```

```

      :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.find(:all, :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 %>

```

```

show.html.erb
html js css
H3 EM [icons] B TD H2 I H1 [icons] BR TR S [icons]

<p>
  <b>Nome:</b>
  <%=h @cliente.nome %>
</p>

<p>
  <b>Idade:</b>
  <%=h @cliente.idade %>
</p>

<%= link_to "Nova qualificação", :controller => "qualificacoes",
      :action => "new",
      :cliente => @cliente %>

<%= link_to 'Edit', edit_cliente_path(@cliente) %> |
<%= link_to 'Back', clientes_path %>
  
```

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 %>
  
```

```

show.html.erb  show.html.erb
html js css
H3 EM [icons] B TD H2 I H1 [icons] BR TR S [icons] js mc js

<%= render :partial => "titulo" %>
<h1>Exibindo Restaurante</h1>

<p>
  <b>Nome: </b>
  <%=h @restaurante.nome %>
</p>

<p>
  <b>Endereco: </b>
  <%=h @restaurante.endereco %>
</p>

<p>
  <b>Especialidade: </b>
  <%=h @restaurante.especialidade %>
</p>

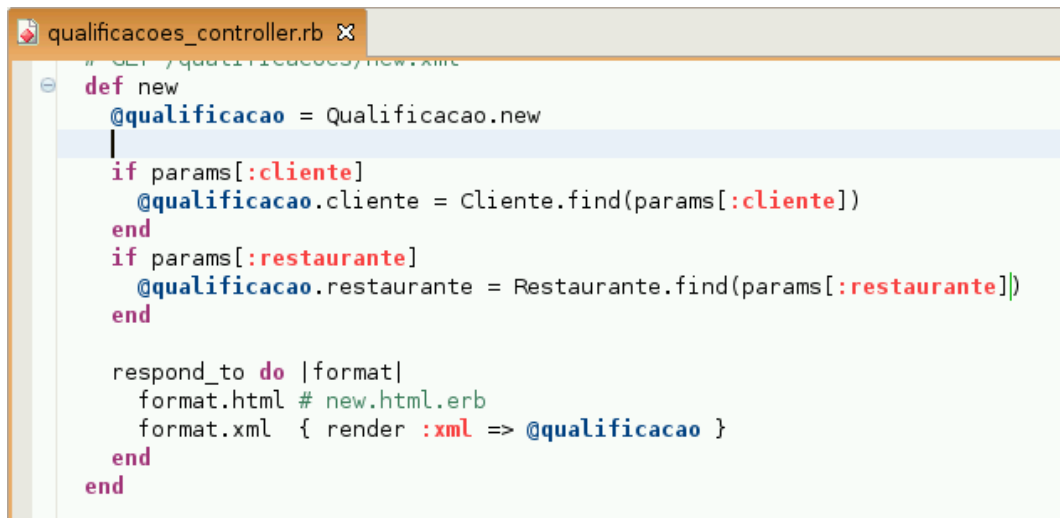
<%= link_to "Nova qualificação", :controller => "qualificacoes",
      :action => "new",
      :restaurante => @restaurante %>

<%= link_to 'Edit', { :action => 'edit', :id => @restaurante } %>
<%= link_to 'Back', { :action => 'index' } %>
  
```

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.

8.4 - Exercícios Opcionais

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)
```

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.

9.1 - 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

9.2 - 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><%=h @restaurante.qualificacoes.average(:nota) %><br/>
```

Podemos mostrar também o número total de qualificações que determinado restaurante possui:

```
<b>Qualificações: </b><%=h @restaurante.qualificacoes.count %><br/>
```

E, por último, fica fácil adicionar o valor médio gasto pelos clientes que visitam tal restaurante:

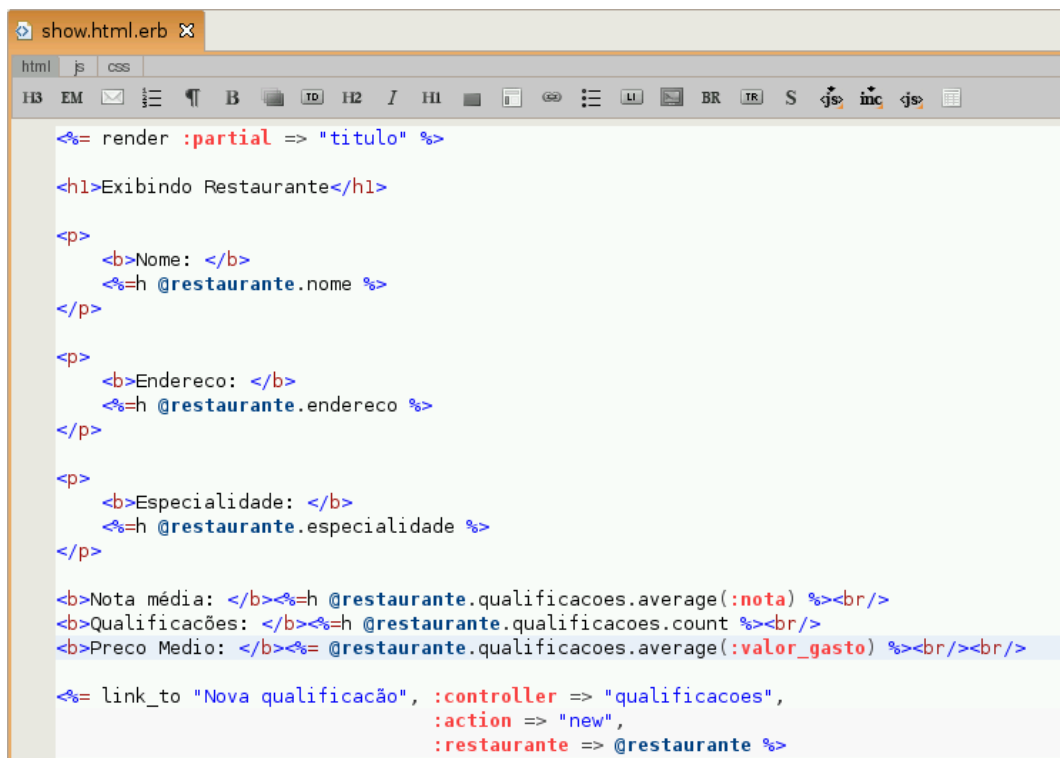
```
<b>Preço médio: </b><%=h @restaurante.qualificacoes.average(:valor_gasto) %><br/>
```


9.3 - 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.rhtml**:

```
<b>Nota média: </b><%=h @restaurante.qualificacoes.average(:nota) %><br/>
<b>Qualificações: </b><%=h @restaurante.qualificacoes.count %><br/>
<b>Preço médio: </b><%=h @restaurante.qualificacoes.average(:valor_gasto) %><br/><br/>
```



```
<%= render :partial => "titulo" %>

<h1>Exibindo Restaurante</h1>

<p>
  <b>Nome: </b>
  <%=h @restaurante.nome %>
</p>

<p>
  <b>Endereco: </b>
  <%=h @restaurante.endereco %>
</p>

<p>
  <b>Especialidade: </b>
  <%=h @restaurante.especialidade %>
</p>

<b>Nota média: </b><%=h @restaurante.qualificacoes.average(:nota) %><br/>
<b>Qualificações: </b><%=h @restaurante.qualificacoes.count %><br/>
<b>Preço Medio: </b><%= @restaurante.qualificacoes.average(:valor_gasto) %><br/><br/>

<%= link_to "Nova qualificação", :controller => "qualificacoes",
  :action => "new",
  :restaurante => @restaurante %>
```

b) Entre no link **http://localhost:3000/restaurantes** e escolha um restaurante para ver suas estatísticas.

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.

10.1 - 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`.

10.2 - 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:

```
script/generate scaffold comentario \  
  conteudo:text comentavel_id:integer comentavel_type:string
```

Caso seja necessário, podemos ainda adicionar índices físicos nas colunas do relacionamento, deixando a migration criada como a seguir:

```
class CreateComentarios < ActiveRecord::Migration  
  def self.up  
    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  
  
  def self.down  
    drop_table :comentarios  
  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  
  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:

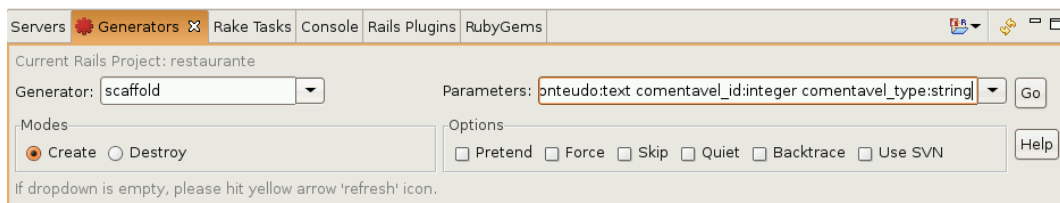
```
class Restaurante < ActiveRecord::Base  
  has_many :qualificacoes  
  
  has_many :comentarios, :as => :comentavel  
  
  # ...  
end
```

A tradução do texto pode ser quase literal: o modelo **TEM MUITOS** comentários **COMO** comentável.

10.3 - Exercícios

1) Vamos criar o modelo do nosso comentário e fazer a migração para o banco de dados:

- Entre na view “Generator”
- Selecione **scaffold**
- Digite: comentario conteudo:text comentavel_id:integer comentavel_type:string
- Aperte “Go”



- Vamos inserir alguns índices físicos. Abra o arquivo **app/db/migrate/<timestamp>_create_comentarios.rb**
- Insira as seguintes linhas:

```
add_index :comentarios, :comentavel_type  
add_index :comentarios, :comentavel_id
```

```
comentario.rb 20080728173953_create_comentarios.rb X
class CreateComentarios < ActiveRecord::Migration
  def self.up
    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

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

g) Na view “Rake Task”, selecione **db:migrate**

h) Aperte “Go”

2) Vamos modificar nossos modelos:

a) Abra o arquivo **app/models/comentario.rb**

b) Adicione a seguinte linha:

```
belongs_to :comentavel, :polymorphic => true
```

```
comentario.rb X
class Comentario < ActiveRecord::Base
  belongs_to :comentavel, :polymorphic => true
end
```

c) Abra o arquivo **app/models/qualificacao.rb**

d) Adicione a seguinte linha:

```
has_many :comentarios, :as => :comentavel
```

```
application.html.erb  qualificacao.rb X
class Qualificacao < ActiveRecord::Base
  belongs_to :cliente
  belongs_to :restaurante

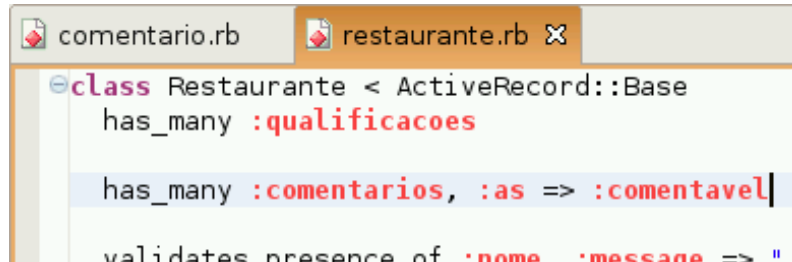
  has_many :comentarios, :as => :comentavel

  validates presence of :nota, :mensagem => " - deve
```

e) Abra o arquivo **app/models/restaurante.rb**

f) Adicione a seguinte linha:

```
has_many :comentarios, :as => :comentavel
```



```
class Restaurante < ActiveRecord::Base
  has_many :qualificacoes

  has_many :comentarios, :as => :comentavel

  validates presence of :nome :message => "
```

3) Para o próximo capítulo, iremos precisar que o nosso sistema já inclua alguns comentários. Para criá-los, você pode usar o script/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.

Ajax fácil com RJS

“O Cliente tem sempre razão”
– Selfridge, H.

Nesse capítulo, você verá como trabalhar com AJAX usando RJS.

11.1 - Adicionando comentários nas views

Até agora trabalhamos com arquivos `html.erb`, que são páginas html com scripts em ruby, mas nada de javascript.

Como Ruby on Rails é um framework voltado para o desenvolvimento web, é natural que a questão do javascript seja levantada. Rails já possui um conjunto de funcionalidades prontas, entre eles os RJS Templates, para facilitar o nosso trabalho com javascript.

11.2 - Métodos de RJS Templates

Falta ainda escrever a funcionalidade de adicionar comentários aos nossos restaurantes e qualificações. Uma sugestão seria utilizar AJAX para uma experiência mais marcante no uso do site pelos usuários.

Para utilizar AJAX temos que definir pedaços de páginas que serão mostrados dinamicamente. Ruby on Rails já vem com suporte embutido às bibliotecas JavaScript **Prototype.js** e **Script.aculo.us**, mas existem plugins para facilitar o trabalho com outras, como JQuery e ExtJS. O primeiro passo é adicionar os javascripts necessários da pasta **public/javascripts** às nossas páginas. Felizmente, existe um helper para nos ajudar nesta tarefa:

```
<%= javascript_include_tag :all %>
```

Ao invés do `:all`, podemos usar também a opção `:defaults`, que não inclui todos os arquivos JavaScript do diretório `public/javascript`; apenas os pré-definidos pelo Rails. Para não precisar copiar e colar esse helper em todas as páginas, o lugar ideal para inseri-lo é na seção `<head>` do nosso layout: **app/views/layouts/application.html.erb**.

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 helper:

```
def comentarios(comentavel)
  comentarios = "<div id='comentarios'>"
  comentarios << "<h3>Comentarios</h3>"
  comentavel.comentarios.each do |comentario|
    comentarios << render(:partial => "comentarios/comentario",
```

```
      :locals => { :comentario => comentario })  
  
    end  
    comentarios << "</div>"  
  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:

```
def comentarios(comentavel)  
  comentarios = "<div id='comentarios'>"  
  comentarios << "<h3>Comentarios</h3>"  
  comentarios << render(:partial => "comentarios/comentario",  
    :collection => comentavel.comentarios) unless comentavel.comentarios.empty?  
  comentarios << "</div>"  
end
```

Agora, vamos criar o partial responsável pela renderização de cada um dos comentários. Repare que usamos o método `link_to_remote`, que faz uma chamada a uma url utilizando para isso uma requisição assíncrona (AJAX):

```
<!-- /app/views/comentarios/_comentario.html.erb -->  
<p id="comentario_<%= comentario.id %>">  
  <%= comentario.conteudo %> -  
  <%= link_to_remote '(remover)', :url => comentario_path(comentario),  
    :method => :delete %>  
</p>
```

Além disso, precisamos invocar nosso helper em **app/views/restaurantes/show.html.erb** e **app/views/qualificacoes/show.html.erb**:

```
<%= comentarios @qualificacao %>
```

Tente clicar no link e verá que nada acontece, porém ao recarregar a página, o comentário foi removido!

A ação `ComentariosController#destroy` está sendo chamada de forma assíncrona (Ajax), porém a página não foi atualizada. Este é o papel dos *RJS templates*: atualizar as páginas de forma assíncrona, na resposta de uma requisição Ajax.

Basta fazermos com que nossa action `destroy` renderize um template RJS, adicionando mais um formato ao bloco `respond_to`:

```
def destroy  
  @comentario = Comentario.find(params[:id])  
  @comentario.destroy  
  
  respond_to do |format|  
  
    format.html { redirect_to(comentarios_url) }  
    format.xml { head :ok }  
    format.js do  
      render :update do |page|  
        page.visual_effect :fade, "comentario_#{params[:id]}"  
      end  
    end  
  end
```



```
end  
end
```

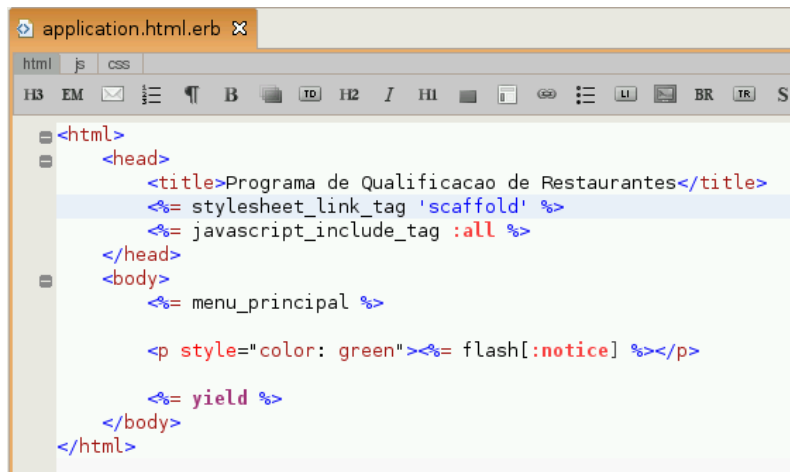
11.3 - Exercícios

1) Vamos adicionar os comentários nas views.

a) Abra o arquivo **app/views/layouts/application.html.erb**

b) Insira a seguinte linha na seção **head**:

```
<%= javascript_include_tag :all %>
```



c) Abra o arquivo **app/helpers/application_helper.rb**

d) Insira as seguintes linhas:

```
def comentarios(comentavel)  
  comentarios = "<div id='comentarios'>"  
  comentarios << "<h3>Comentarios</h3>"  
  comentarios << render(:partial => "comentarios/comentario",  
                        :collection => comentavel.comentarios)  
  comentarios << "</div>"  
end
```

```

application.html.erb  application_helper.rb
# Methods added to this helper will be available to all templates in the application.
module ApplicationHelper
  def menu_principal
    @menu = %w(cliente qualificacao restaurante)
    menu_principal = "<ul>"
    @menu.each do |item|
      menu_principal << "<li>" + link_to(item, :controller => item.pluralize) + "</li>"
    end
    menu_principal << "</ul>"
    menu_principal
  end

  def comentarios(comentavel)
    comentarios = "<div id='comentarios'>"
    comentarios << "<h3>Comentários</h3>"
    comentarios << render(:partial => "comentarios/comentario",
                        :collection => comentavel.comentarios)
    comentarios << "</div>"
  end

  def valor_formatado(number)
    number_to_currency(number, :unit => "R$", :separator => ",", :delimiter => ".")
  end
end

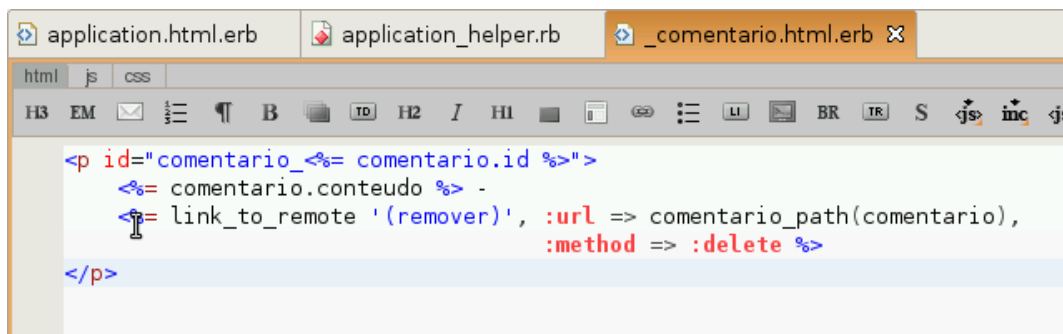
```

e) Crie o arquivo **app/views/comentarios/_comentario.html.erb** com o seguinte conteúdo:

```

<p id="comentario_<%= comentario.id %>">
  <%= comentario.conteudo %> -
  <%= link_to_remote '(remover)', :url => comentario_path(comentario),
    :method => :delete %>
</p>

```



```

application.html.erb  application_helper.rb  _comentario.html.erb
html  js  css
H3  EM  [icons]  B  TD  H2  I  H1  [icons]  BR  TR  S  [icons]  inc  [icons]

<p id="comentario_<%= comentario.id %>">
  <%= comentario.conteudo %> -
  <%= link_to_remote '(remover)', :url => comentario_path(comentario),
    :method => :delete %>
</p>

```

f) Adicione as seguintes linhas à action **destroy** do controller **comentarios_controller.rb**

```

format.js do
  render :update do |page|
    page.visual_effect :fade, "comentario_#{params[:id]}"
  end
end

```

```

show.html.erb | show.html.erb | comentarios_controller.rb x
# DELETE /comentarios/1
# DELETE /comentarios/1.xml
def destroy
  @comentario = Comentario.find(params[:id])
  @comentario.destroy

  respond_to do |format|
    format.html { redirect_to(comentarios_url) }
    format.xml { head :ok }
    format.js do
      render :update do |page|
        page.visual_effect :fade, "comentario_#{params[:id]}"
      end
    end
  end
end
end
end
  
```

g) Adicione a seguinte linha em **app/views/restaurantes/show.html.erb**

```
<%= comentarios @restaurante %>
```

h) Adicione a seguinte linha em **app/views/qualificacoes/show.html.erb**

```
<%= comentarios @qualificacao %>
```

```

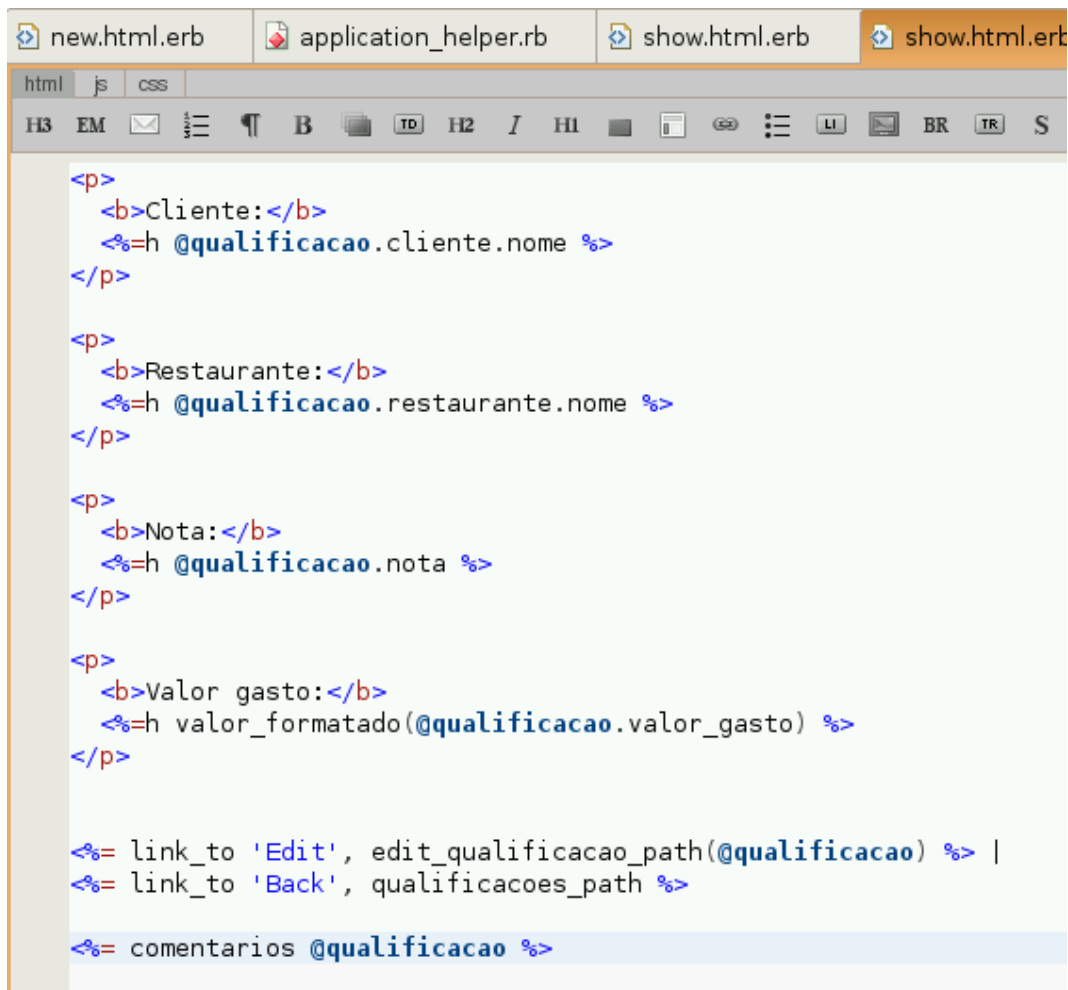
show.html.erb x | show.html.erb
html | js | css
H3 EM [icons] B TD H2 I H1 [icons] LI [icons] BR TR
<p>
  <b>Cliente:</b>
  <%=h @qualificacao.cliente.nome %>
</p>

<p>
  <b>Restaurante:</b>
  <%=h @qualificacao.restaurante.nome %>
</p>

<p>
  <b>Nota:</b>
  <%=h @qualificacao.nota %>
</p>

<p>
  <b>Valor gasto:</b>
  <%=h valor_formatado(@qualificacao.valor_gasto) %>
</p>

<%= link_to 'Edit', edit_qualificacao_path(@qualificacao) %> |
<%= link_to 'Back', qualificacoes_path %>
  
```



```
<p>
  <b>Cliente:</b>
  <%=h @qualificacao.cliente.nome %>
</p>

<p>
  <b>Restaurante:</b>
  <%=h @qualificacao.restaurante.nome %>
</p>

<p>
  <b>Nota:</b>
  <%=h @qualificacao.nota %>
</p>

<p>
  <b>Valor gasto:</b>
  <%=h valor_formatado(@qualificacao.valor_gasto) %>
</p>

<%= link_to 'Edit', edit_qualificacao_path(@qualificacao) %> |
<%= link_to 'Back', qualificacoes_path %>

<%= comentarios @qualificacao %>
```

- i) Teste em <http://localhost:3000/restaurantes>, escolhendo um restaurante no qual você já tenha inserido comentários.

11.4 - Adicionando comentários

Para possibilitar a adição de comentários podemos adicionar um novo link logo após a lista de comentários, editando nosso helper method:

```
def comentarios(comentavel)
  comentarios = "<div id='comentarios'>"
  comentarios << "<h3>Comentarios</h3>"
  comentarios << render(:partial => "comentarios/comentario",
                       :collection => comentavel.comentarios)
  comentarios << "</div>"
  comentarios << render(:partial => "comentarios/novo_comentario",
                       :locals => { :comentavel => comentavel })
end
```

É necessário agora criar o novo partial. Lembre que para criar um comentário, é preciso saber o que está sendo comentado (comentável).

```
<!-- /app/views/comentarios/_novo_comentario.html.erb -->
<div id='novo_comentario'>
  <%= link_to_remote 'Novo comentário', :update => 'novo_comentario',
    :url => new_comentario_path(
      'comentario[comentavel_type]' => comentavel.class,
      'comentario[comentavel_id]' => comentavel) %>
</div>
```

A opção `:update`, diz qual o id do elemento que será atualizado com a resposta da requisição ajax. Bastante útil quando apenas um elemento da página é atualizado.

Após isso, precisamos modificar a action “new” do controlador dos comentários para receber os parâmetros passados. Para isso, basta passarmos o parâmetro na chamada ao “new”:

```
@comentario = Comentario.new(params[:comentario])
```

A inserção de novos comentários já deve estar funcionando, porém o formulário de comentários mostra os campos “Comentavel” e “Comentavel type” para serem editados. Não faz sentido fazer com que o usuário tenha que digitá-los, portanto podemos mudá-los para campos hidden, no formulário de comentários:

```
<%= f.hidden_field :comentavel_type %>
<%= f.hidden_field :comentavel_id %>
```

Aproveite para remover os labels destes campos também e o link de ‘Voltar’ no final da página.

11.5 - Exercícios

1) Vamos permitir a adição de novos comentários:

- Abra o arquivo **app/helpers/application_helper.rb**
- Adicione as seguintes linhas ao final do método “**comentarios**”:

```
comentarios << render(:partial => "comentarios/novo_comentario",
  :locals => { :comentavel => comentavel })
```

```

application_helper.rb
# Methods added to this helper will be available to all templates in the application.
module ApplicationHelper
  def menu_principal
    @menu = %w(cliente qualificacao restaurante)
    menu_principal = "<ul>"
    @menu.each do |item|
      menu_principal << "<li>" + link_to(item, :controller => item.pluralize) + "</li>"
    end
    menu_principal << "</ul>"
    menu_principal
  end

  def comentarios(comentavel)
    comentarios = "<div id='comentarios'>"
    comentarios << "<h3>Comentários</h3>"
    comentarios << render(:partial => "comentarios/comentario",
                        :collection => comentavel.comentarios)
    comentarios << "</div>"
    comentarios << render(:partial => "comentarios/novo_comentario",
                        :locals => { :comentavel => comentavel })
  end

  def valor_formatado(number)
    number_to_currency(number, :unit => "R$", :separator => ",", :delimiter => ".")
  end
end

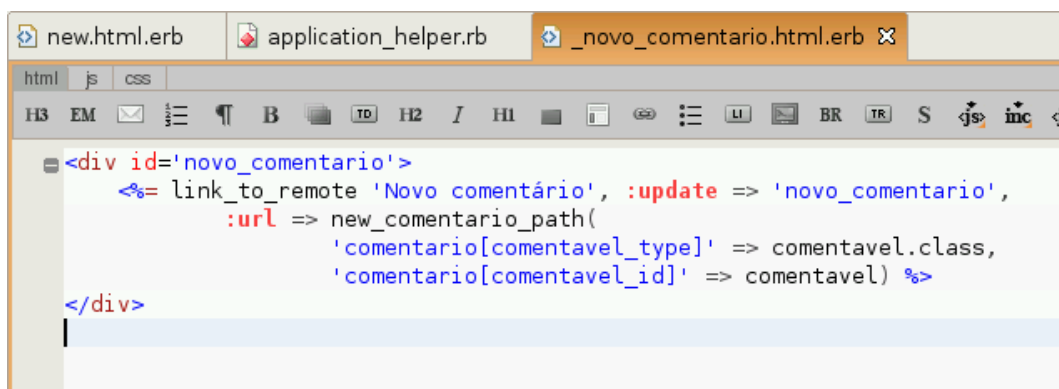
```

c) Crie o novo partial **app/views/comentarios/_novo_comentario.html.erb** com o seguinte conteúdo:

```

<div id='novo_comentario'>
  <%= link_to_remote 'Novo comentário', :update => 'novo_comentario',
    :url => new_comentario_path(
      'comentario[comentavel_type]' => comentavel.class,
      'comentario[comentavel_id]' => comentavel) %>
</div>

```



d) Abra o arquivo **app/views/comentarios/new.html.erb**

e) Troque as linhas:

```

<p>
  <%= f.label :comentavel_id %><br/>
  <%= f.text_field :comentavel_id %>
</p>
<p>
  <%= f.label :comentavel_type %><br/>

```

```

    <%= f.text_field :comentavel_type %>
  </p>
por

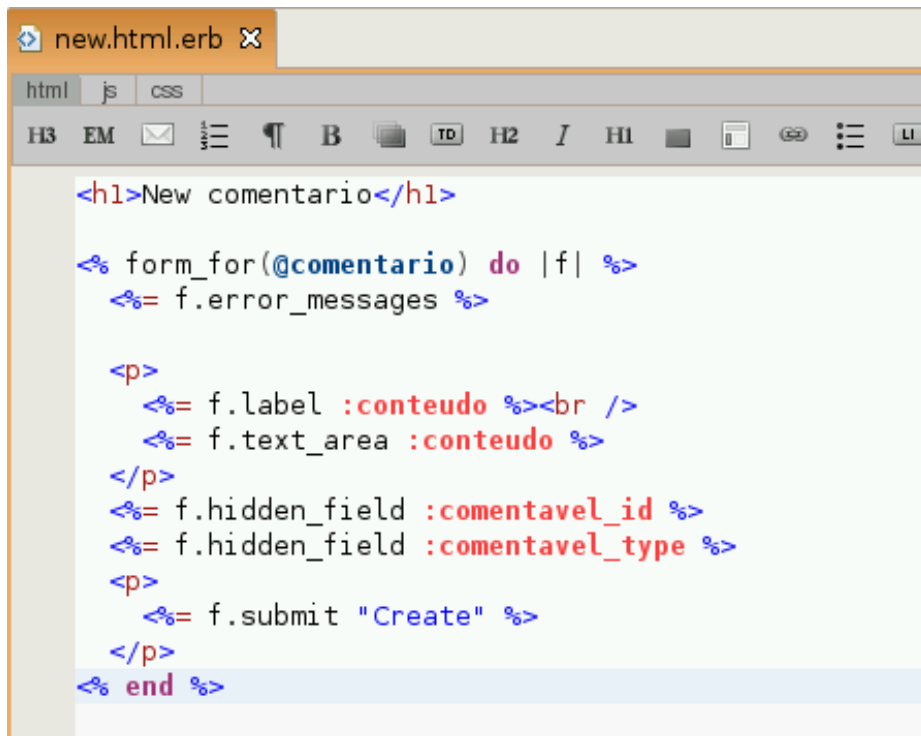
```

```

    <%= f.hidden_field :comentavel_id %>
    <%= f.hidden_field :comentavel_type %>

```

f) Aproveite para remover o link de 'Voltar' no final da página.



```

new.html.erb
html js css
H3 EM [icons] B TD H2 I H1 [icons] LI

<h1>New comentario</h1>

<%= form_for(@comentario) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :conteudo %><br />
    <%= f.text_area :conteudo %>
  </p>
  <%= f.hidden_field :comentavel_id %>
  <%= f.hidden_field :comentavel_type %>
  <p>
    <%= f.submit "Create" %>
  </p>
<%= end %>

```

g) Precisamos também fazer com que a action new receba os parâmetros passados. Edite o arquivo **app/controller/comentarios_controller.rb** e troque as linhas:

```

def new
  @comentario = Comentario.new

  respond_to do |format|
    format.html # new.html.erb
    format.xml { render :xml => @comentario }
  end
end

por

def new
  @comentario = Comentario.new(params[:comentario])

  respond_to do |format|
    format.html # new.html.erb
    format.xml { render :xml => @comentario }
  end
end

```

```
# GET /comentarios/new
# GET /comentarios/new.xml
def new
  @comentario = Comentario.new(params[:comentario])

  respond_to do |format|
    format.html # new.html.erb
    format.xml { render :xml => @comentario }
  end
end
```

h) Teste em <http://localhost/restaurantes>, adicionando um novo comentário

11.6 - Exercícios - Enviando os dados com Ajax

- 1) Para submeter os dados de forma assíncrona, basta trocar o helper de formulário para usar a versão Ajax, disponível em `ActionViewHelpersPrototypeHelper`.

```
<!-- app/views/comentarios/new.html.erb -->
<% remote_form_for(@comentario) do |f| %>
  <p>
    <b>Conteúdo</b><br />
    <%= f.text_area :conteudo %>
  </p>

  <%= f.hidden_field :comentavel_type %>
  <%= f.hidden_field :comentavel_id %>

  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>
```

A submissão já funciona, mas a página não é atualizada. Uma das formas é responder a action `create`, que salva o comentário, com um RJS. Adicione um novo formato suportado pela action `create`:

```
if @comentario.save
  # ...
  respond_to do |format|
    format.js
    format.html { redirect_to(@comentario) }
    # ...
  end
end
```

Ao invés de renderizar o rjs direto no controlador, podemos usar a view de resultado `app/views/comentarios/create.js.rjs`:

```
# app/views/comentarios/create.js.rjs
page.visual_effect :shake, 'novo_comentario'
page.visual_effect :puff, 'novo_comentario'
```



```
page.insert_html :bottom, 'comentarios',
  :partial => 'comentario',
  :locals => { :comentario => @comentario }

page.insert_html :after, 'comentarios',
  :partial => 'novo_comentario',
  :locals => { :comentavel => @comentario.comentavel }

page.visual_effect :highlight, "comentario_#{@comentario.id}"
```

Alguns Plugins e Gems Importantes

“A minoria pode ter razão, a maioria está sempre errada”
— Mikhail Aleksandrovitch Bakunin

12.1 - Paginação

O uso do plugin `will_paginate` sempre foi recomendado ao invés da forma clássica de paginação embutida no Rails 1.2.

No Rails 2.0, a paginação foi removida para o plugin `classic_pagination` e agora você deve escolher qual usar.

Hoje o `will_paginate` é um gem. Para poder usá-lo nas suas aplicações Rails:

```
gem install will_paginate
```

A documentação pode ser encontrada no wiki hospedado no `github.com`:

http://github.com/mislav/will_paginate/wikis

Após a instalação do gem, para usá-lo em uma aplicação Rails, basta inserir no final do arquivo `config/environment.rb`:

```
require 'will_paginate'
```

Para trazer os dados paginados, basta usar o método `paginate`, no lugar de `find`:

```
@restaurantes = Restaurante.paginate :page => params['page']
```

O método `paginate` funciona como um `finder` normal. Suporta todas as opções previamente vistas, como `:conditions`, `:order` e `:include`.

O número de itens por página é padronizado em 30, mas você pode customizar de duas formas. Através do método `per_page` nas classes `ActiveRecord::Base`:

```
class Restaurante < ActiveRecord::Base
  def self.per_page
    10
  end
  # ...
end
```

Ou passando como parâmetro ao método `paginate`:

```
@restaurantes = Restaurante.paginate :page => params['page'], :per_page => 10
```

O gem fornece ainda um helper para inclusão dos links de paginação em qualquer view:

```
# index.html.erb
<% @restaurantes.each do |restaurante| %>
  <li><%= restaurante.nome %></li>
<% end %>
<%= will_paginate @restaurantes %>
```

Alguns exemplos de estilos (css) alternativos para a paginação podem ser encontrados em:

http://mislav.caboo.se/static/will_paginate/

12.2 - Exercícios - Título

- 1) No terminal, instale a gem do **will_paginate**.

```
gem install will_paginate
```

- 2) Abra o arquivo **app/controllers/restaurantes_controller.rb**. Na action list, troque a linha:

```
@restaurantes = Restaurantes.find(:all)
por
```

```
@restaurantes = Restaurantes.paginate :page=> params['page'], :per_page=>3
```

- 3) Abra o arquivo **config/environment.rb**. Adicione a linha abaixo na última linha do arquivo, depois do "end".

```
require 'will_paginate'
```

- 4) Abra o arquivo **app/views/restaurantes/index.html.erb**. Adicione a linha abaixo após a tabela:

```
<%= will_paginate @restaurantes %>
```

- 5) Abra a listagem de restaurantes e verifique a paginação.

12.3 - Hpricot

Hpricot 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 hpricot
gem install open-uri
```

open-uri é um gem que usaremos para fazer requisições http:

```
doc = Hpricot(open('http://twitter.com/fabiokung'))
```

Analisando o html gerado pelo twitter, vemos que as mensagens estão sempre dentro de elementos com a classe "hentry"

. Além disso, dentro de cada `hentry`, a única parte que nos interessa é o conteúdo dos subitens de classe `"entry-content"`

.

Podemos procurar estes itens com o `Hpricot`, usando seletores `CSS`. Expressões `XPath` também poderiam ser usadas:

```
doc / ".hentry .entry-content"
```

Para imprimir cada um dos itens de uma maneira mais interessante:

```
items = doc / ".hentry .entry-content"
items.each do |item|
  puts item.inner_text
end
```

12.4 - Exercícios - Testando o Hpricot

1) Vamos fazer um leitor de twitties de um determinado usuário

- Crie um arquivo chamado `"twitter_reader.rb"`
- Adicione às seguintes linhas:

```
require 'hpricot'
require 'open-uri'

doc = Hpricot(open('http://twitter.com/caueguerra'))
items = doc / ".hentry .entry-content"
items.each do |item|
  puts item.inner_text
end
```

- Teste usando o comando `ruby twitter_reader.rb`

12.5 - 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") { |f| f.write(params[:upload]['picture_path'].read) }
```

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 modelos `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 `ActiveRecord`. As configurações são simples e é possível validar tamanho do arquivo ou tornar sua presença obrigatória. O paperclip tem como pré-requisito o `ImageMagick`,

12.6 - Exercícios

- 1) Para instalar o paperclip, abra o arquivo config/environment.rb e adicione a gem:

```
config.gem 'thoughtbot-paperclip', :lib => 'paperclip', :source => 'http://gems.github.com'
```

E no terminal, rode :

```
rake gems:install
```

```
rake gems:unpack
```

- 2) 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 Rail buscará pelo thumb.

```
class Restaurante < ActiveRecord::Base
  has_attached_file :foto, :styles => { :medium => "300x300>", :thumb => "100x100>" }
end
```

- 3) 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.

Crie a migration **AddFotoColumnsToRestaurante** abaixo na pasta db/migrate:

```
class AddFotoColumnsToRestaurante < ActiveRecord::Migration
  def self.up
    add_column :restaurantes, :foto_file_name, :string
    add_column :restaurantes, :foto, :string
    add_column :restaurantes, :foto_file_size, :integer
    add_column :restaurantes, :foto_updated_at, :datetime
  end

  def self.down
    remove_column :restaurantes, :foto_file_name
    remove_column :restaurantes, :foto
    remove_column :restaurantes, :foto_file_size
    remove_column :restaurantes, :foto_updated_at
  end
end
```

Rode a migration no termina com:

```
rake db:migrate
```

- 4) Abra a view **app/views/restaurantes/new.html.erb** e altere o formulário. Seu form deve ficar como o abaixo:

```
<% form_for :restaurantes, :url => {:action=>'create'}, :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.

Apêndice A - Testes

“Ninguém testa a profundidade de um rio com os dois pés.”

– Provérbio Africano

13.1 - O Porquê dos testes?

Testes Unitários 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

Extreme Programming é um conjunto de práticas de programação que visam a simplicidade, praticidade, qualidade e flexibilidade de seu sistema. Os testes unitários fazem parte dessa metodologia de programação.

O Ruby já possui classes que nos auxiliam no desenvolvimento destes testes.

13.2 - `Test::Unit`

`Test::Unit` é a biblioteca usada para escrever suas classes de teste.

Ao escrever testes em Ruby utilizando esse framework, você deve estender a classe `TestCase` que provê a funcionalidade necessária para fazer os testes.

```
require 'test/unit'
```

```
class PessoaTest < Test::Unit::TestCase
  # ...
end
```

Ao estender `Test::Unit::TestCase`, você herda 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::Unit::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.

No RadRails, temos a própria view de testes, basta clicar no botão “Run all Tests” e o RadRails abrirá a view de testes unitários.

Ao clicar no botão “Run all tests”, o RadRails executa todos os arquivos que encontrar no diretório **test/** da aplicação Rails e que possuírem o sufixo **_test.rb**. Também é possível executar todos os testes com algumas tasks do rake:

```
rake test           # roda todos os testes unitários, de integração e funcionais
rake test:units     # roda todos os testes da pasta test/unit
rake test:functionals # roda todos os testes da pasta test/functional
rake test:integration # roda todos os testes da pasta test/integration
rake test:plugins   # roda todos os testes de plugins, na pasta vendor/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 unitários 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 unitários
- `assert_redirected_to(path)` - para testes de controladores
- `assert_template(esperado)` - também para controladores
- entre outros

13.3 - RSpec

Muito mais do que uma nova forma de criar testes unitários RSpec fornece uma forma de criar especificações executáveis do seu código.

No TDD, descrevemos a funcionalidade esperada para nosso código através de testes unitários. BDD (*Behavior Driven Development*) leva isso ao extremo e diz que nossos testes unitários 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:

```
gem install rspec-rails
cd projeto-rails
script/generate rspec
```

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://github.com/dchelimsky/rspec-rails/wikis/home>

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 idéias 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>?.`

```
objeto.should be_empty      # testa: objeto.empty?
objeto.should_not be_nil    # testa: not objeto.nil?
```

```
objeto.should be_kind_of(Restaurante) # testa: objeto.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
```

```
objeto.should eql(outro) # testa: objeto.eql?(outro)
objeto.should equal(outro) # testa: objeto.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' }
itens.should have_key(:dois) # testa: itens.has_key?(:dois)
itens.should_not have_value(/3/) # testa: not itens.has_value?(/3/)
```

- `be_close`, inclui tolerância.

```
conta = 10.0 / 3.0
conta.should be_close(3.3, 0.1) # == 3.3`.1
```

- `have(num).<colecão>`, para testar a quantidade de itens em uma associação.

```
categoria.should have(15).produtos # testa categoria.produtos.size == 15
```

Um uso especial deste *matcher* é para objetos que já são coleções. Neste caso, podemos usar o nome que quisermos:

```
array = [1,2,3]
array.should have(3).items # testa array.size == 3
array.should have(3).numbers # mesma coisa
```

- `have_at_least(num).<colecão>`: mesma coisa que o anterior, porém usa `>=`.
- `have_at_most(num).<colecão>`: mesma coisa que o anterior, porém usa `<=`.
- `match`, para expressões regulares.

```
texto.should match(/^F/) # verifica se começa com F
```

Este são os principais, mas ainda existem outros. Você pode encontrar a lista de Matchers completa na documentação do módulo `Spec::Matchers`:

<http://rspec.rubyforge.org/rspec/1.1.11/classes/Spec/Matchers.html>

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
```

13.4 - 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 rspec rspec-rails cucumber webrat
```

Para projetos rails, é possível usar o *generator* fornecido pelo Cucumber para adicionar os arquivos necessários ao projeto:

```
cd projetorails  
ruby script/generate cucumber
```

As User Stories são chamadas de **Features** pelo Cucumber. São arquivos de texto puro com a extensão *.feature*. Arquivos com definição de features sempre contêm uma descrição da funcionalidade (**Story**) e alguns exemplos (**Scenários**), 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 incluía 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 (**Scenários**) que explicam a funcionalidade. Cada um dos Scenários 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 condicao descrita no arquivo texto puro" do
  # codigo a ser executado para este passo
end
```

```
Given /e outra condicao com valor: (.*)/ do |valor|
  # codigo 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*.

Apêndice B - 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.

14.1 - 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 estender 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.

14.2 - 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:

14.3 - Exercícios

1) Crie um arquivo chamado testando.rb que imprime na tela “Testando o JRuby!”:

- Edite o arquivo: testando.rb.
- Adicione o seguinte conteúdo:

```
print "Testando o JRuby!\n"
```


c) Rode o arquivo com o JRuby:

```
jruby testando.rb
```

14.4 - 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"
obj = java.lang.Object.new
```

A chamada de métodos é feita da mesma maneira que em Java, utilizando o operador “.” (ponto):

```
obj.to_string
```

Agora observe o conteúdo abaixo:

```
require 'java'
module Swing
  include_package 'java.awt'
  include_package 'javax.swing'
end

module AwtEvent
  include_package 'java.awt.event'
end

frame = Swing::JFrame.new
label = Swing::JLabel.new
label.text = "Testando o JRuby!"
# label.setText("Testando o JRuby!")

frame.add(label)
frame.setSize(400, 400)
frame.visible = true
```

O `include_package` é parecido com um `import`, e depois estamos criando uma instância de `JFrame`. Para quem conhece swing e java, é um dos containers de componentes de visualização.

Dessa mesma maneira você pode acessar qualquer outra classe da biblioteca do Java. Ou mais ainda: se quiser pode usar uma biblioteca java externa, como o hibernate ou JFreeChart.

Assim você tem toda a expressividade e poder do Ruby, somado a quantidade enorme de bibliotecas do Java.

Apêndice C - Deployment

— Alguem

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.

15.1 - 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
ruby script/server
```

Por padrão, o Webrick inicia na porta 3000, porém isto pode ser mudado com a opção `-p`:

```
ruby script/server -p 3002
```

Caso o Mongrel (outro servidor famoso, descrito nas próximas seções) esteja instalado, tem prioridade sobre o Webrick no comando `script/server`. Podemos forçar a execução do webrick com:

```
script/server webrick
```

Por ser muito simples, **não é recomendado** o uso do webrick em produção.

15.2 - 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.

15.3 - 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.

15.4 - 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>

15.5 - 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.

15.6 - 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 idéia 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**.

15.7 - 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 protocolo **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>

15.8 - 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.

15.9 - 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!

Apêndice D - Instalação

“A dúvida é o principio da sabedoria”
— Aristóteles

16.1 - Ruby - Ubuntu

Para instalar o Rails no Ubuntu deve-se executar os seguintes passos:

- Build essencial para instalação de gems com extensão nativa:

```
sudo apt-get install build-essential
```

- Instalar ruby

```
sudo apt-get install ruby1.8 ruby1.8-dev
```

- Precisamos também instalar o rubygems. Existe um pacote **deb** no **apt-get**, mas que é melhor não usar pois ele está sempre desatualizado. Então vamos fazer o download e instalação manuais. Faça o download da última versão:

http://rubyforge.org/frs/?group_id=126 (1.1.1)

Basta descompactar e executar no diretório:

```
sudo ruby setup.rb
```

Depois de instalar, não se esqueça de atualizar o rubygems:

```
sudo gem update --system
```

Feito isso, você já estará com o Ruby e rubygems mais atuais.

O Ubuntu cria os executáveis com o sufixo 1.8. Para tudo funcionar como esperado, é necessário ainda criar alguns links:

```
sudo ln -s /usr/bin/gem1.8 /usr/bin/gem
sudo ln -s /usr/bin/ruby1.8 /usr/bin/ruby
sudo ln -s /usr/bin/rdoc1.8 /usr/bin/rdoc
```




```
sudo ln -s /usr/bin/ri1.8 /usr/bin/ri
sudo ln -s /usr/bin/irb1.8 /usr/bin/irb
```

16.2 - Ruby - Windows

Para instalar no Windows, basta fazer o download da última versão do ruby em:

http://rubyforge.org/frs/?group_id=167.

Basta descompactar e dar duplo clique para instalar.

Após isso, é necessário atualizar a rubygem. Digite no prompt de comando:

```
gem update --system
```

16.3 - Rails

Para instalar Rails, basta executar o seguinte comando:

No Ubuntu:

```
sudo gem install rails
```

No Windows, abra um prompt de comando e digite:

```
gem install rails
```

16.4 - JDK

Para utilização do editor Aptana, é necessária a instalação da JDK.

No Ubuntu:

```
sudo apt-get install sun-java6-jdk
```

No Windows:

Fazer o download em <http://java.sun.com/javase/downloads/index.jsp>

16.5 - Aptana

O Aptana Studio pode ser adquirido em: <http://www.aptana.com/studio/download>

Após fazer o download do Aptana, basta descompactá-lo e clicar em "Aptana". Feito isso, é necessário instalar o plugin para manipulação de projetos Rails. Para isso, faça:



Menu Help -> Software Updates -> Find and Install -> Search for new features to install

Selecione Aptana: RadRails Development Environment -> next O Aptana irá fazer download o plugin Rails automaticamente

16.6 - Mongrel

Para instalar o mongrel, basta executar o seguinte comando:

No Ubuntu:

```
sudo gem install mongrel
```

No Windows, abra um prompt de comando e digite:

```
gem install mongrel
```

16.7 - MySQL

No Ubuntu:

```
sudo apt-get install mysql-server-5.0
```

No Windows:

Basta fazer o download em <http://dev.mysql.com/downloads/mysql/5.0.html>

Após isso, basta descompactar e dar duplo clique para instalar.

16.8 - SVN

No Ubuntu:

```
sudo apt-get install subversion
```

No Windows:

Basta fazer o download em:

<http://subversion.tigris.org/servlets/ProjectDocumentList?folderID=91>

Após isso, basta dar duplo clique para instalar.

Índice Remissivo

ActiveRecord, 48

after_{filter}, 92

around_{filter}, 93

BDD, 135

before_{filter}, 92

Behavior Driven Development, 135

Boolean, 10

Builder, 91

Classes abertas, 15

CoC, 30

Comparable, 27

Copy on Write, 147

DRY, 30

Duck Typing, 26

Enumerable, 27

ERb, 79

gems, 5

Gemstone, 7

HAML, 92

hpricot, 129

if, 10

IronRuby, 6

JRuby, 6

Maglev, 7

MetaProgramming, 21

Migrations, 50

mod_{rails}, 7

mongrel_{cluster}, 146

MRI, 4

MSpec, 6

MVC, 30

open-uri, 129

Operadores Aritméticos, 8

Operadores Booleanos, 10

ORM, 48

Phusion, 7

Phusion Passenger, 7

RadRails, 31

Rake, 49

Regexp, 11

respond_{to}, 98

RSpec, 135

Rubinius, 6

Ruby, 4

Ruby Enterprise Edition, 7

Ruby.NET, 6

RubyGems, 5

RubySpec, 6

scaffold, 37

Search Engine Optimization, 95

SEO, 95

String, 9

Symbol, 10

Syntax Sugar, 17

tcmalloc, 147

Testes, 133

The Great Ruby Shootout, 7

will_{paginate}, 128