

Capítulo

3

Introdução ao Desenvolvimento de Aplicações Web com Ruby on Rails

Daniel Cárnio Junqueira, Renata Pontin de Mattos Fortes

Abstract

*This chapter presents the concepts of the framework **Ruby on Rails** and the use of them during a web application development. The development of a web application is detailed, as a practical illustration of development of the typical web applications, using for example relational databases. Additionally, the described development is based on an agile process. The differential topic introduced is related to techniques of version control on distributed repositories, including identification of developer's roles (programmers and designers).*

Resumo

*Neste capítulo são apresentados os conceitos do framework **Ruby on Rails** e sua utilização para o desenvolvimento de aplicações web. Assim, o desenvolvimento de uma aplicação web é discutido em detalhes, de forma a servir de exemplo prático de desenvolvimento de aplicações web típicas, que utilizam, por exemplo, bancos de dados relacionais. Além disso, o desenvolvimento é apresentado com base no processo ágil. O diferencial introduzido é a utilização de técnicas de controle de versões com repositórios distribuídos, com identificação das responsabilidades dos desenvolvedores (programadores e designers).*

3.1. Introdução

Com o avanço da Internet, é possível observar um movimento de migração de aplicações *Desktop* para aplicações *web*, além da criação de novos tipos de aplicações que não existiam no mundo sem Internet, como as aplicações de comércio eletrônico e serviços governamentais oferecidos pela rede e diversas outras [Jazayeri, 2007].

Neste contexto, também evoluíram as tecnologias e técnicas para o desenvolvimento de aplicações *web*. Inicialmente, a maior parte das aplicações era desenvolvida em linguagem Perl ou como *scripts* CGI [Lerner, 1997] desenvolvidos em linguagem C e outras linguagens procedurais, sem utilização do paradigma de Orientação a Objetos. Posteriormente, foram desenvolvidas linguagens bastante voltadas para o desenvolvimento de aplicações *web*, como PHP [Riddell, 2000]. Entretanto, foi apenas após o ano 2000, época em que o acesso à Internet começou a se popularizar mais em todo o mundo, que começaram a ser desenvolvidas soluções com o objetivo de resolver os principais problemas existentes no desenvolvimento deste tipo de aplicações: as aplicações *web* são extremamente dinâmicas, não guardam estado, utilizam um protocolo simples para comunicação (HTTP [Fielding et al., 1999]) que, por definição, permite apenas tipos simples de interação e que deve ser iniciada sempre do lado do cliente, entre outros. A Figura 3.1 mostra a arquitetura básica de uma aplicação *web*, destacando as características de comunicação do protocolo HTTP.

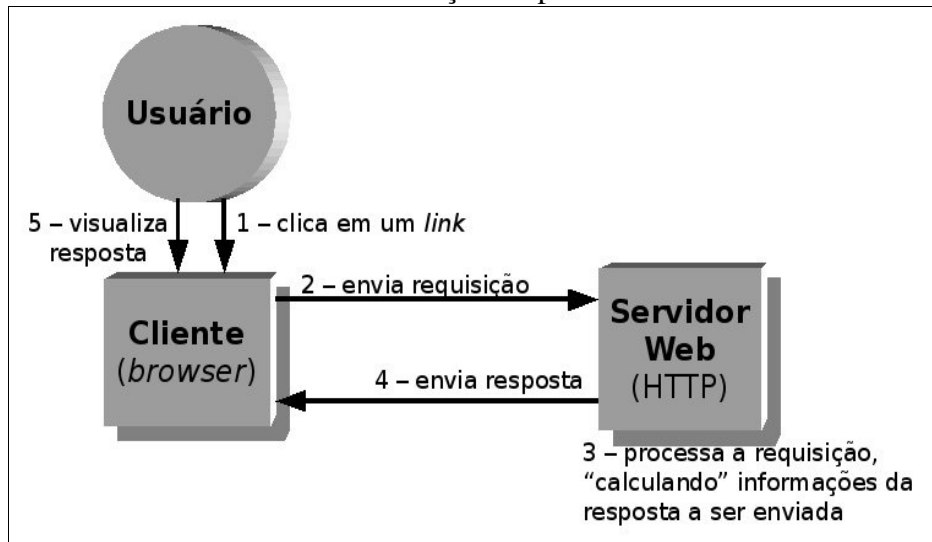


Figura 3.1: Arquitetura básica das Aplicações Web e detalhes de Comunicação entre as máquinas cliente-servidor envolvidas, com base no Protocolo HTTP

Pode-se notar pela ordem das ações representadas na Figura 3.1, que o protocolo HTTP não permite que a comunicação seja iniciada no servidor, ou seja, o funcionamento do servidor é sempre relacionado ao processamento de uma requisição (*HTTP Request* – ação 2 na Figura 3.1) feita pelo lado cliente, que gera então uma resposta (*HTTP Response* – ação 4 na Figura 3.1) que é enviada novamente para o *browser*.

Nas aplicações *Desktop*, mesmo quando desenvolvidas numa arquitetura cliente-servidor, é dada ao desenvolvedor mais liberdade na comunicação entre a interface gráfica que é exibida para o usuário com a parte responsável pelo tratamento das informações e controle da interface, sendo que a comunicação pode ser iniciada a partir dos dois lados. Dessa forma, podem ser observados tanto os eventos do usuário como os eventos que estão ocorrendo nos dados e então qualquer notificação necessária pode ser feita para a interface gráfica.

Para melhorar a interação com o usuário nas aplicações *web*, várias tecnologias continuam surgindo e evoluindo, como *Ajax* [Stamey e Richardson, 2006] *JavaFX*¹, *AdobeFlex*². Já para melhorar a organização da arquitetura de aplicações *web* no servidor, vários *frameworks* têm sido propostos: *Struts*, *Spring*, *CakePHP*, *RIFE*, *Microsoft.NET (ASP.NET)*, entre outros. Estes *frameworks* têm o objetivo de auxiliar o desenvolvimento e a manutenção de aplicações voltadas para a *web*.

Observa-se que, atualmente, as tecnologias predominantes nas principais aplicações *web* são da Sun (Java) e Microsoft (plataforma .NET). Os principais *frameworks* para Java são *Struts*, *Java Server Faces*, *Hibernate* (persistência), *Spring*, e outros, utilizados por grandes empresas que desenvolvem as aplicações utilizando a plataforma *Java Enterprise Edition* (JEE). A Microsoft propõe uma solução completa em sua plataforma .NET .

Entretanto, todas as tecnologias são relativamente novas, e um grupo de desenvolvedores de software começou a contestar as soluções abertas feitas em Java por serem muitas vezes complicadas de utilizar, com muitas configurações para serem feitas, bem como a solução da Microsoft devido a questões comerciais. Com o objetivo de simplificar o desenvolvimento de aplicações *web*, começaram o desenvolvimento de um *framework* completamente novo, numa linguagem aberta – Ruby -, o que resultou no **Ruby on Rails**, que foi lançado em meados de 2004.

Neste capítulo, é apresentado o *framework* **Ruby on Rails**, oferecendo-se os elementos introdutórios aos conceitos subjacentes. Além disso, são apresentados os conceitos básicos sobre controle de versões com repositórios distribuídos e separação de papéis no desenvolvimento de aplicações *web*, que é possível de ser realizado utilizando-se **Ruby on Rails**.

O capítulo está organizado da seguinte forma: na Seção 3.2 é apresentada a linguagem de programação Ruby; na Seção 3.3 é detalhado o *framework* **Ruby on Rails** descrevendo os principais conceitos; na Seção 3.4 é apresentado um exemplo de aplicação desenvolvida com **Ruby on Rails** destacando os aspectos do ambiente de

1 <http://www.adobe.com/br/products/flex/>

2 <http://java.sun.com/javafx/>

desenvolvimento, com separação de papéis para as atividades de desenvolvimento; na Seção 3.5 são apresentadas as perspectivas do desenvolvimento de aplicações web.

3.2 A Linguagem Ruby

Ruby é uma linguagem de programação que foi desenvolvida no Japão por Yukihiro “Matz” Matsumoto. Seu desenvolvimento foi iniciado em 1993, e ela foi lançada em 1995. Tanto a especificação da linguagem como a implementação dos interpretadores e outras ferramentas são distribuídos gratuitamente [Lerner, 2006] .

Ruby é uma linguagem bastante simples e flexível, por vezes vista como uma combinação de Perl e Smalltalk. Suas principais características são:

- Orientada a Objetos: tudo em Ruby é um objeto – não existe nenhum tipo primitivo. Em Ruby, mesmo os números e o valor nulo são objetos;
- Tipagem dinâmica: as variáveis não têm seu tipo declarado de forma explícita. Ele é atribuído dinamicamente de acordo com o valor assumido pela variável;
- Variáveis dinâmicas: as variáveis não precisam ser declaradas e seu tipo é dinâmico;
- Suporte a pacotes, mix-ins, entre outros
- Interpretada: é uma linguagem interpretada, o que permite rápido desenvolvimento pelo fato de não ser necessário realizar os passos de compilação e implantação (*deploy*).

A Figura 3.2 contém a lista de todas as palavras reservadas da linguagem **Ruby**, enquanto a Figura 3.3 contém todos os seus operadores.

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

Figura 3.2: Palavras reservadas da Linguagem Ruby

**	!	~	*	/	%	+	-	&
<<	>>	 	^	>	>=	<	<=	<=>
 	!=	=~	!~	&&	+=	--	==	===
..	...	not	and	or				

Figura 3.3: Operadores da Linguagem Ruby

Uma meta de Matz era reduzir a sobrecarga do desenvolvedor para seguir o princípio de evitar surpresas (*Principle Of Least Surprise* - POLS), ou seja, de que a linguagem oferecesse o que o desenvolvedor espera sem efeitos ou consequências inesperadas, assim como por exemplo, nomear métodos com termos usuais de ações em inglês relativas ao que se quer executar.

Em relação à sintaxe de Ruby, ela combina características da sintaxe de Perl com as de Smalltalk, e apresenta semelhanças com Python e Lisp. Um exemplo de um arquivo de código-fonte (oiPessoa.rb) na Linguagem Ruby seria:

```
1 def oiPessoa(nome)
2   resultado = "Ola, " + nome
3   return resultado
4 end
5 puts oiPessoa("Abigail")
```

Para rodar esse programa, basta chamar na linha de comando:

```
> ruby oiPessoa.rb
```

Como resultado de sua execução, será apresentado: Ola, Abigail

Este exemplo ilustra, de maneira simples, alguns elementos da linguagem. Na primeira linha, está sendo definido um método da aplicação 'Oi Pessoa'. Os métodos correspondem às ações que devem ser executadas. Este método recebe como argumento o parâmetro 'nome'. Na segunda linha, uma variável de instância recebe o valor resultante da concatenação da palavra 'Ola, ' com o argumento 'nome'.

Dev-se notar que diferente de outras linguagens, Ruby não exige que seja definido o tipo da variável, pois ele é determinado com base no valor que a variável recebe. Esta característica confere à Ruby a qualificação de ser uma linguagem com tipagem dinâmica. Na terceira linha, o valor de resultado simplesmente é retornado para quem chamou o método. A quarta linha indica que o método está completo. Todos os métodos devem terminar com 'end'.

Finalmente, a última linha é a ação que realmente 'oiPessoa.rb' vai executar. O método *puts* é um método padrão de Ruby para mostrar um valor na tela.

Este mesmo exemplo poderia ser reescrito da seguinte forma, no arquivo oiPessoa2.rb:

```
1 class Cumprimento
2   def initialize (nome)
3     puts oiPessoa(nome)
4   end
5   def oiPessoa(nome)
6     resultado = "Ola #{nome}!"
7   end
8 end
9 cumprimento = Cumprimento.new("Abigail")
```

Para rodar esse programa, basta chamar na linha de comando:

```
> ruby oiPessoa2.rb
```

Neste segundo exemplo, vários outros aspectos da linguagem Ruby podem ser percebidos. Em primeiro lugar, é definida uma classe “Cumprimento”, que contém os possíveis métodos de cumprimentos. Note que o nome da classe inicia com uma letra maiúscula: esta regra é imposta pela linguagem. Além disso, na linha 2 é definido o método “initialize”: este é o método construtor da classe.

Também deve ser dada atenção à linha 6: o 'nome' é incluído na *string* na forma de `#{nome}` – esta técnica é chamada de *interpolação de strings*, e está presente também em outras linguagens de programação alto-nível. Este tipo de inclusão de variáveis em *strings* só pode ser realizado nas *strings* definidas com aspas duplas (“.”). Em Ruby, aspas simples (‘.’) também podem ser utilizadas para definir *strings*, mas seu conteúdo é tratado de forma literal, não sendo possível a interpolação de variáveis.

Outro aspecto que deve ser notado neste exemplo é que no método definido entre as linhas 5 e 7 não existe a expressão “return”, mas o resultado é retornado quando o método é invocado. Esta é outra característica da linguagem Ruby – em qualquer método, o resultado do último ítem processado no método é retornado.

A linguagem Ruby suporta também herança simples entre classes. O exemplo a seguir ilustra como funciona a herança em Ruby:

```
# - arquivo pessoa.rb -----
1 class Pessoa
2   attr_accessor :nome
3   def initialize (nome)
4     @nome = nome
5   end
6 end
# - arquivo estudante.rb -----
1 require "pessoa"
2 class Estudante < Pessoa
3   attr_accessor :nota
4 end
```

No exemplo acima apresentado, é possível notar mais algumas características da linguagem Ruby. Primeiramente, explorando o conteúdo do arquivo “pessoa.rb”:

- na linha 2 - “attr_accessor :nome” - é utilizado o método “attr_accessor”, que é definido na classe de sistema Module. Ruby fornece três métodos para a criação de *accessors* (métodos *setters* e *getters*): **attr_reader**, que cria apenas métodos para recuperação do atributo, **attr_writer**, que cria o método “atributo=” para gravação do valor, e **attr_accessor**, que cria tanto o método para recuperação quanto o método para escrita do atributo (*getters* e *setters*). Este método é bastante útil e oferece os mecanismos de encapsulamento em Ruby – não é possível acessar os atributos caso não sejam gerados os *getters* e *setters*. Ainda na linha 2, é utilizado um símbolo: “:nome”. Símbolos são como os identificadores simples (nome de variável, por exemplo), mas são precedidos por dois pontos (:). Eles são *strings* que são automaticamente convertidas em constantes – é criado um objeto em memória que será referenciado pelo nome do símbolo em qualquer parte do programa Ruby.
- na linha 4, é criada uma variável de instância, com a utilização do símbolo “@”. Esta variável possui um valor único para cada vez que a classe Pessoa é instanciada num objeto. Além deste tipo de variável, Ruby permite a criação de variáveis de classe, que são precedidas por duas arrobas (@@), e variáveis globais, precedidas por um \$.

Já no arquivo estudante.rb, podemos notar a presença dos seguintes elementos da linguagem Ruby:

- na linha 1 – “require pessoa” – a palavra reservada “require” serve para incluir outra classe na classe atual. Neste caso, como nossa classe Estudante será uma especialização de Pessoa, é necessário chamar o método “**require**”.
- na linha 2 – símbolo “<”: – indica qual a classe que está sendo especializada pela classe declarada neste arquivo. Neste caso, a classe Estudante é uma especialização de Pessoa, e, portanto, herda os seus métodos e atributos, inclusive o método construtor (**initialize**).

Um arquivo teste.rb pode mostrar a utilização das classes acima criadas:

```
1 ignacio = Estudante.new("Ignacio Luiz")
2 ignacio.nota= 3
3 pedro = Pessoa.new("Pedro Pedroso")
4 puts "#{ignacio.nome} #{ignacio.nota}"
5 puts "#{pedro.nome} #{pedro.nota}"
```

Quando este arquivo de testes for executado, a seguinte saída será mostrada na tela:

Ignacio Luiz 3

```
NoMethodError: undefined method `nota' for #<Pessoa:0xb7ce42b4 @nome="Pedro Pedroso">
```

Como é possível observar, serão impressos corretamente os dados para o objeto “ignacio” mas, quando solicitamos a impressão do atributo “nota” para o objeto “pedro”, é gerado um erro, pois pedro é uma instância de “Pessoa”, que não possui o atributo “nota” nem seu *getter* e *setter*.

3.3 Ruby on Rails

O **Rails** ou **Ruby on Rails** é um *framework* de código aberto para desenvolvimento de aplicações web, implementado em **Ruby** [Williams, 2007]. Ele foi criado por **David Heinemeier Hansson**, na Dinamarca, e foi publicado em julho de 2004. Este *framework* foi projetado tendo em vista os seguintes objetivos:

- ser uma solução de desenvolvimento completa, fornecendo num único *framework* todas as ferramentas necessárias para a criação de aplicações web, incluindo persistência, lógica e apresentação;
- permitir a comunicação entre suas camadas da forma mais transparente possível;
- ser uniforme, escrito totalmente apenas numa linguagem;
- seguir o padrão de arquitetura **MVC** (*Model-View-Controller*).

Estes objetivos, explícitos no site do projeto (<http://www.rubyonrails.org/>), foram definidos para que se pudesse desenvolver um *framework* que fosse produtivo e que mantivesse baixa a curva de aprendizagem. Conforme relatado pela equipe de desenvolvimento, realmente conseguiram atingir a maioria dos objetivos, e o conjunto de características de **Rails** possibilita que o desenvolvedor se sinta bastante produtivo, mantendo baixa a curva de aprendizagem.

Com a mesma intenção de aumentar a produtividade do desenvolvedor, dois conceitos são seguidos pelo *framework*: **DRY** (*Don't Repeat Yourself*) e *Convention over Configuration*, os quais são brevemente compreendidos como a seguir:

- **DRY – Don't Repeat Yourself** é o conceito por trás da técnica de definir nomes, propriedades e códigos em somente um lugar, reaproveitando esta informação em outros lugares que seja necessário. Por exemplo, ao invés de ser mantida uma tabela “papers” e uma classe “Paper”, com uma propriedade, um *getter* e um *setter* para cada campo na tabela, esta informação é mantida apenas em banco de dados. Todas as propriedades e métodos necessários são “injetados” na classe através de funcionalidades do *framework*. Isso gera uma grande economia de tempo e aumenta a manutenibilidade das aplicações desenvolvidas – alterando

apenas o banco de dados, tudo o que se baseia nestas informações é atualizado automaticamente

- **Convention over Configuration** – conceito de que deve-se assumir valores padrão onde existe uma convenção. Ao desenvolvedor é dada a liberdade de fazer alterações conforme sua vontade, de forma que não siga a convenção, mas, se seguir a convenção, geralmente nenhuma configuração será necessária, e o tempo de desenvolvimento será reduzido. Por exemplo, por convenção, os dados da classe “Paper” devem ficar na tabela “papers”; caso isto seja feito, não será necessária nenhuma configuração. Porém, se o desenvolvedor quiser, pode colocá-los em outra tabela, mas deverá configurar a aplicação para indicar essa informação. Neste caso, deve ser feita uma observação em relação ao idioma utilizado: por padrão, o *framework* possui mapeadas as regras para pluralização do idioma inglês. Mais adiante nesta seção são explicadas as possíveis soluções no caso de desenvolvimento de aplicações em português, inclusive com nomenclatura das entidades em português.

Na verdade, **Ruby on Rails** é formado por um conjunto de cinco (5) outros *frameworks*, a saber: *Action Mailer*, *Action Pack*, *Action Web Service*, *Active Record* e *Active Support*. A seguir, uma breve descrição de suas funções:

1. **Action Mailer** é um *framework* para o desenvolvimento de camadas que contenham serviços de *e-mail*. Estas camadas são utilizadas para criar código para enviar senhas perdidas, mensagens de boas vindas durante a assinatura de determinado site ou serviço, e quaisquer outros casos de uso que requeiram uma notificação escrita a uma pessoa ou sistema. Além disso, uma classe *Action Mailer* pode ser utilizada para processar mensagens de email recebidas, como por exemplo permitir que um weblog receba postagens diretamente do email. Este *framework* não é detalhado neste capítulo.
2. **Action Pack** é responsável por fazer a separação da resposta de uma requisição web em uma parte de controle (que faz a lógica) e outra de visão (processando um template). Esta abordagem é realizada em dois passos, para normalmente criar (*create*), ler (*read*), atualizar (*update*) ou deletar (*delete*) (CRUD – *create, read, update, delete*) a parte do modelo, que geralmente tem um banco de dados por trás, antes de escolher entre renderizar um template ou redirecionar para outra *action*. Neste *framework* são implementados o *Action Controller* e a *Action View*. Este *framework* é mais detalhado ao longo do capítulo.
3. **Action Web Service** fornece mecanismos para publicar APIs de *web services* interoperacionais com Rails sem dispendar muito tempo aprofundando nos detalhes de protocolo. Este *framework* não é detalhado neste capítulo.

4. **Active Record** é responsável por fazer o mapeamento entre os objetos da aplicação e as tabelas da base de dados para criar um modelo de persistência do domínio onde a lógica e os dados são apresentados e tratados de uma única forma. É uma implementação do padrão (*design pattern*) *Active Record*. Este *framework* é detalhado neste capítulo.
5. **Active Support** é uma coleção de várias classes utilitárias e extensões de bibliotecas padrão que foram consideradas úteis para **Rails**. Todas estas bibliotecas facilitadoras de desenvolvimento foram empacotadas juntas. Embora não seja feito um detalhamento explícito das facilidades disponibilizadas por este *framework* neste capítulo, algumas de suas características são apresentadas.

Além de o *framework* ser escrito na linguagem **Ruby**, todas as aplicações desenvolvidas com o *framework* também devem ser desenvolvidas nesta linguagem. Conforme será apresentado mais adiante, mesmo os *templates* utilizam uma variação da linguagem, batizada de *eRuby*.

3.3.1. O Padrão de Projeto MVC

O *framework* **Ruby on Rails**, além de ter sido desenvolvido com o objetivo de aumentar a produtividade e manter baixa a curva de aprendizagem, também foi pensado como um *framework* que incentiva as boas práticas de codificação e organização das aplicações. Neste âmbito, deve ser destacada a arquitetura das aplicações desenvolvidas sobre Rails, que seguem o Padrão de Projeto (*design pattern*) MVC: *Model-View-Controller* [Sasine e Toal, 1995].

MVC é um padrão de projeto no qual as aplicações são desenvolvidas em camadas (Figura 3.4). Neste caso, existem as camadas de modelo, visão e controle. Em cada uma dessas camadas, deve-ser alocar diferentes partes da aplicação em desenvolvimento, conforme o seguinte critério:

- **Modelo** – representação das informações operadas pela aplicação. Por exemplo, **author**, **title** e **abstract** fazem parte do domínio de um sistema para revisão sistemática. Esta camada opera com os dados crus, e não possui a lógica necessária para o tratamento dos dados. Em **Ruby on Rails**, esta camada é manipulada pelo *Active Record*, que é uma implementação do *design pattern* *Active Record*.
- **Visão** – camada responsável por exibir os dados (modelo) em uma forma específica para a interação – no caso do Ruby on Rails, páginas web disponibilizadas no navegador. **Rails** fornece o *framework* *Action View* para compor a camada de visão.

- **Controle** – a camada de controle é responsável por processar e responder aos eventos, podendo invocar alterações na camada de modelo. Em **Ruby on Rails**, o *Action Controller* possui as funcionalidades para esta camada.

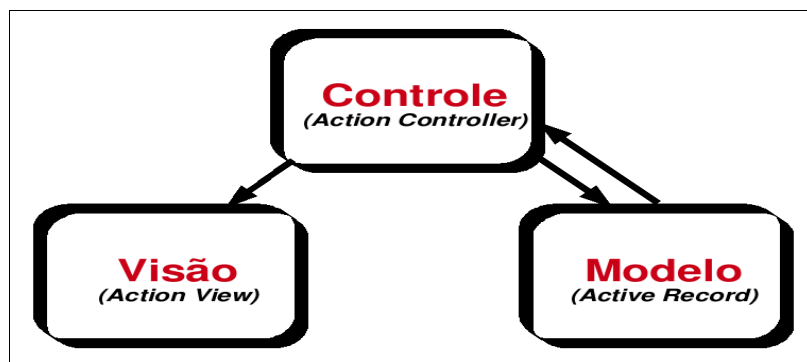


Figura 3.4: Esquema das camadas do padrão MVC

O fluxo de eventos que ocorrem entre as camadas do MVC é o seguinte:

1. O usuário interage com a interface (botões, links);
2. O *Controller* manipula o evento da interface, através de uma rotina pré-escrita, também conhecida como *action*;
3. O *Controller* acessa o *Model*, atualizando-o caso seja necessário;
4. A *View* utiliza o *Model* para gerar uma interface apropriada, recuperando os dados atualizados;
5. A interface do usuário espera por próximas interações, que iniciarão o ciclo novamente (conforme já foi descrito anteriormente, o protocolo HTTP não permite que uma comunicação seja iniciada no servidor).

3.3.2. Conteúdo das aplicações em Ruby on Rails

O *framework* **Ruby on Rails** tem como principal objetivo auxiliar os desenvolvedores de aplicações *web*, reduzindo ao máximo o trabalho manual e a repetição de informações em diferentes locais.

Com o objetivo de organizar as aplicações desenvolvidas sobre o *framework*, a seguinte estrutura de diretórios é utilizada:

- **app**: diretório que contém todo o código específico da aplicação;
- **app/controllers**: armazena os *controllers*. A nomenclatura destes arquivos deve ser, por exemplo, `weblogs_controller.rb` para que seja feito o mapeamento de URLs de forma automática. Todos os *controllers* devem ser derivados de **ApplicationController**, que por sua vez é derivado de **ActionController::Base**;

- **app/models:** contém os modelos, que devem seguir a nomenclatura como, por exemplo, `post.rb`. A maioria dos modelos irá ser derivado de `ActiveRecord::Base`;
- **app/views:** armazenam os arquivos de *template* que devem seguir a nomenclatura como `weblogs/index.rhtml` para a *action* `WeblogsController#index`. Todos os arquivos desta camada utilizam a sintaxe *eRuby*;
- **app/views/layouts:** contém os arquivos de *template* dos *layouts* que são utilizados com as *views*. Estes *layouts* devem conter, por exemplo, os cabeçalhos e rodapés comuns a todas as páginas ou a um determinado grupo de páginas;
- **app/helpers:** contém *helpers* para a *view* que devem seguir a nomenclatura como `weblogs_helper.rb`. Estes arquivos são gerados automaticamente quando é utilizado o “gerador” de um *controller*. Os *helpers* podem ser utilizados para empacotar funcionalidades das *views* em métodos;
- **config:** arquivos de configuração para o ambiente **Rails**, o mapa de roteamento, o banco de dados e outras dependências;
- **components:** mini-aplicações auto-contidas que podem ser distribuídas junto com *controllers*, *models* e *views*;
- **db:** contém o esquema do banco de dados no arquivo `schema.rb`;
- **db/migrate:** contém a seqüência de *Migrations* para o esquema;
- **doc:** diretório onde será armazenada a documentação da aplicação, quando a mesma é gerada automaticamente com a ferramenta **rake doc:app**;
- **lib:** bibliotecas específicas da aplicação. Basicamente, qualquer tipo de código customizado que não pertence aos controladores, modelos ou *helpers*. Este diretório é carregado automaticamente (está no *path* de carregamento);
- **public:** o diretório disponível para o servidor web. Contém subdiretórios para imagens, folhas de estilo e javascripts. Também contém os *dispatchers* e os arquivos HTML default;
- **script:** contém *scripts* auxiliares para automação e geração;
- **test:** testes de unidade e funcionais. Quando os scripts de geração são utilizados, arquivos de teste de modelo são criados e colocados neste diretório automaticamente.

A Figura 3.5 mostra a estrutura de diretórios de uma aplicação desenvolvida em **Ruby on Rails** que foi criada automaticamente pela IDE Aptana. Esta aplicação de exemplo foi denominada “systematic_review”.

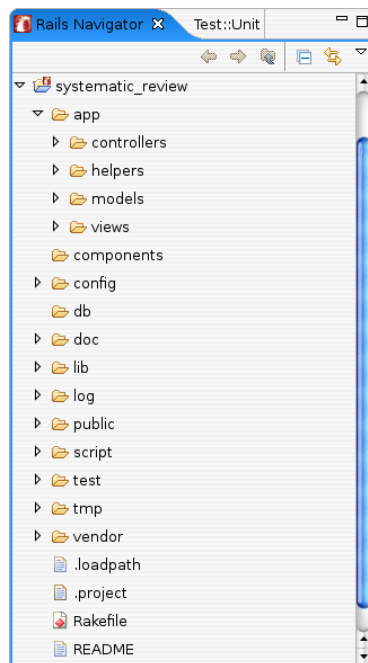


Figura 3.5: Estrutura de diretórios das aplicações em Ruby on Rails

3.3.3 Configurações das aplicações

Ruby on Rails foi concebido como um *framework* que não necessita de muitas configurações caso seja seguida uma convenção. Além disso, seu principal objetivo é suportar o desenvolvimento de aplicações *web* que possuem um sistema de banco de dados no *back-end*. Portanto, é necessário que seja feita a configuração do banco de dados a ser acessado por uma determinada aplicação.

Para se fazer esta configuração, é necessário editar o arquivo **config/database.yml**, exibido na Figura 3.6, que possui os seguintes parâmetros que podem ser configurados, para o ambiente de desenvolvimento, testes e produção:

- **adapter:** indica qual o SGBD que está sendo utilizado. **Rails** fornece adaptadores para diversos SGBDs utilizados atualmente, como MySQL, PostgreSQL, Oracle, entre outros;
- **database:** é o nome do banco de dados utilizado por esta aplicação;
- **username:** nome de usuário que a aplicação utilizará para se conectar;
- **password:** senha a ser utilizada para se conectar;
- **host:** host da máquina que contém o servidor do banco de dados.

É importante salientar que neste arquivo podem ser configurados os ambientes de desenvolvimento, testes e produção. Deve ser tomado cuidado com a configuração do banco de dados de testes, pois este banco será apagado e reconstruído como um novo banco a cada vez que forem executados os *scripts* de testes.

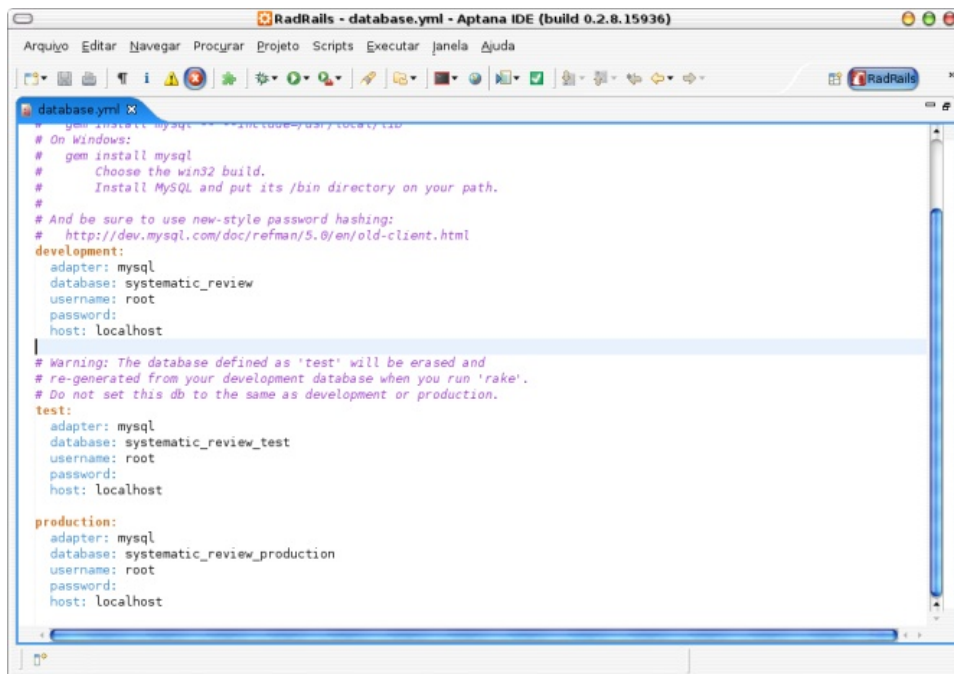


Figura 3.6: Arquivo database.yml

No diretório `/config/environments`, existem os arquivos **development.rb**, **production.rb** e **test.rb**, que possuem, respectivamente, as configurações dos ambientes de desenvolvimento, produção e testes. Geralmente não é necessário alterar as configurações padrão destes arquivos. Além desses arquivos, existe o arquivo **config/environment.rb**, com configurações sobre o ambiente, e o arquivo **config/routes.rb**, no qual podem ser configuradas regras de mapeamento de URL.

Por padrão, **Ruby on Rails** faz a “pluralização” de alguns termos para a camada de modelo – aliás, pela convenção, os dados de “Paper” ficam na tabela “papers” no banco de dados. Entretanto, estas regras não estão disponibilizadas em português no pacote de instalação – apenas as regras de pluralização em inglês são disponibilizadas. Para corrigir esta lacuna, existem duas possíveis soluções:

- desabilitar as regras de pluralização: para desabilitar as regras de pluralização, o arquivo **config/environment.rb** deve ser editado, e, na última linha, deve ser inserido o conteúdo: `ActiveRecord::Base.pluralize_table_names=false`. Quando esta solução for adotada, as tabelas do banco de dados devem ter os nomes no singular.

- instalar um *plugin* que mapeia as regras de pluralização de português, disponível para *download* em <http://www.reinventar.com/2006/06/portuguese-plural-rules-plugin/> (visitado em 19/08/2007)

No arquivo **config/routes.rb**, podem ser configurados mapeamentos de URL em *actions* caso não seja seguido o padrão. Por padrão, os *controllers* são armazenados no diretório **app/controller** e possuem a nomenclatura no formato <entidade>_controller.rb. Além disso, as classes de controle precisam herdar da classe ActionController::Base. As *actions* são criadas como métodos das classes de controle. Na Figura 3.8 é ilustrado o arquivo padrão **config/routes.rb**, e na Figura 3.7 é ilustrado um exemplo de um *controller* de uma aplicação desenvolvida em **Ruby on Rails**. Os *controllers* serão explorados mais adiante neste capítulo, juntamente com outras classes que compõem as aplicações construídas sobre **Ruby on Rails**.

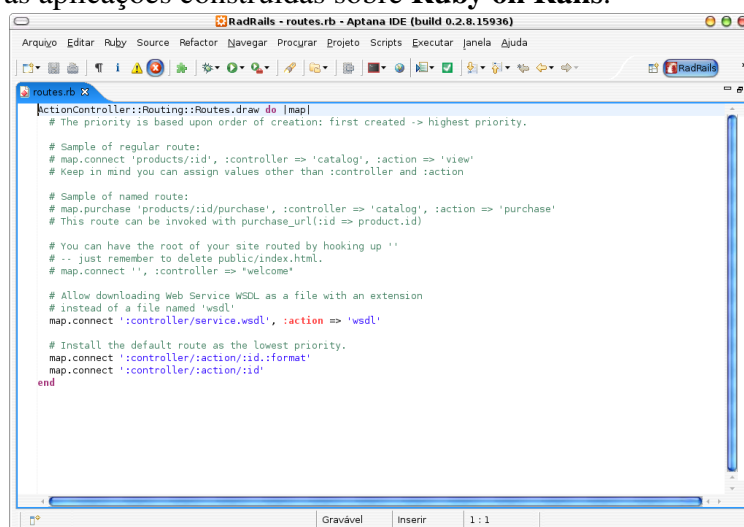


Figura 3.7: Arquivo config/routes.rb

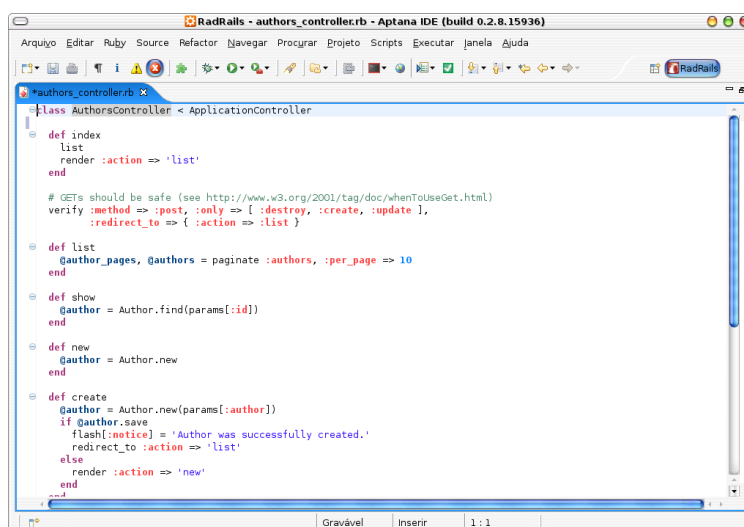


Figura 3.8: Exemplo de *controller* de aplicação em Ruby on Rails

3.3.4 Scripts do Ruby on Rails

Para o *framework* **Ruby on Rails** facilitar o desenvolvimento de aplicações *web*, ele disponibiliza uma série de *scripts* para a geração automática de código, testes, distribuição das aplicações, entre outros.

Os *scripts* disponibilizados ficam no diretório **script**. Os principais *scripts* que fazem parte de qualquer aplicação **Rails** são os seguintes:

- *console*: disponibiliza um console para interação com o ambiente da aplicação Rails, no qual é possível interagir com o modelo do domínio. Neste console, são disponibilizadas todas as partes da aplicação configurada, exatamente igual à quando a aplicação está rodando. É possível inspeção dos modelos, mudar seus valores e salvar os resultados no banco de dados.
- *destroy*: permite “destruir” determinado componente da aplicação, como por exemplo um modelo ou uma *view* que tenham sido criados de forma incorreta ou que não sejam mais necessários
- *generate*: permite que seja feita a geração automática de código. Os geradores padrão do **Rails** são: *controller*, *integration_test*, *mailer*, *migration*, *model*, *observer*, *plugin*, *resource*, *scaffold*, *scaffold_resource*, *session_migration*, *web_service*. Os geradores *scaffold*, *controller* e *model* são explorados em detalhes neste capítulo.
- *server*: permite rodar um servidor básico para a aplicação. Este servidor é bastante utilizado em ambiente de desenvolvimento, por permitir rápido acompanhamento da aplicação. Quando a aplicação for distribuída, pode ser implantada em outros servidores HTTP que aceitem CGI ou FastCGI (preferível).

3.3.5 Criação de aplicações em Ruby on Rails

O roteiro básico para iniciar a criação de aplicações web com **Ruby on Rails** é ilustrado na Figura 3.9.

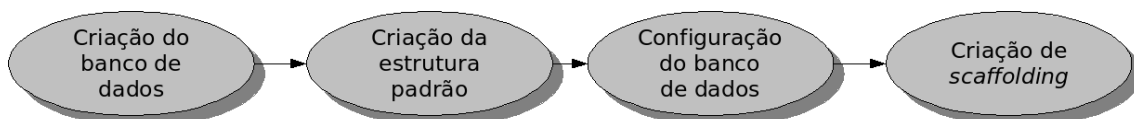


Figura 3.9: Passos para a criação de Aplicações Web com Ruby on Rails

Seguindo a proposta de ordem dos passos apresentada na Figura 3.9, o roteiro para criação de aplicações com **Ruby on Rails** pode ser realizado, considerando-se:

- **Criação do banco de dados:** esta etapa consiste em criar o banco de dados da aplicação. Com isso, deve ser feito o mapeamento do modelo conceitual da aplicação para um banco de dados relacional.
- **Criação da estrutura padrão:** após instalado o framework na máquina de desenvolvimento, o comando “rails” pode ser utilizado para criar uma aplicação. A utilização deste comando é da seguinte forma: rails <nome_da_aplicação>. Desta forma, será criada uma aplicação contendo a estrutura padrão de qualquer aplicação desenvolvida em Rails. Além de poder ser criada com o comando “rails”, a IDE Aptana, indicada neste capítulo, auxilia o trabalho de criação inicial da aplicação. Basta solicitar a criação de um novo projeto na IDE para que a estrutura padrão da aplicação seja gerada.
- **Configuração do banco de dados:** após realizados os passos de criação do banco de dados e criação da estrutura padrão, deve ser configurado o arquivo **config/database.yml** para conter a base de dados de desenvolvimento. Os dados de conexão devem ser informados neste arquivo. Nesta etapa, devem ser informados os dados do banco de desenvolvimento.
- **Criação de *scaffolding*:** é sugerida a utilização de *scaffolding* para agilizar a criação das aplicações. Após configurado o banco de dados, pode-se utilizar o *script* de geração para se criar um *scaffold* da aplicação, da seguinte forma: *script/generate scaffold papers*. Como resultado disso, são criados o modelo, visão e controle para os artigos da aplicação de revisão sistemática tomada como exemplo, e as operações básicas de inserção, visualização, edição e listagem de artigos já passam a ser disponibilizadas. Os arquivos de modelo podem, então, ser editados. Este passo é ainda mais detalhado nesta seção.

As aplicações desenvolvidas em **Ruby on Rails** devem ter seus arquivos organizados segundo a estrutura sugerida pelo *framework*, na qual os arquivos específicos da aplicação são colocados abaixo do diretório **app**. Além disso, as camadas de modelo, visão e controle devem ter seus arquivos separados nos respectivos diretórios.

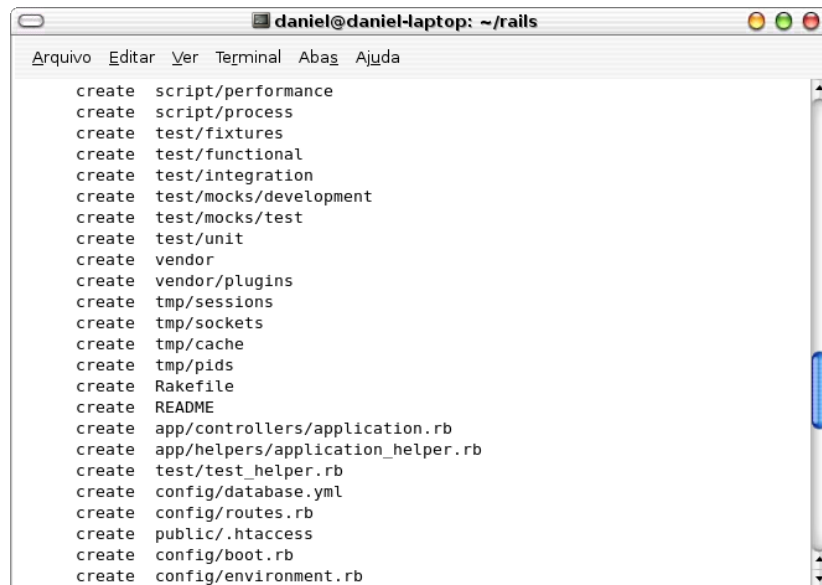
Para o desenvolvimento de aplicações com tal organização, **Ruby on Rails** disponibiliza um gerador automático de código bastante útil, o denominado *scaffold*. O que este gerador faz é criar arquivos de “exemplo” para aplicação, mas que são funcionais o suficiente para permitir a execução das ações de criação, edição, visualização, listagem e exclusão de registros das tabelas criadas no banco de dados. Como exemplo, supõe-se que uma tabela capaz de armazenar os dados de artigos acadêmicos seja criada num banco de dados MySQL com o seguinte script:

```
CREATE TABLE papers (  
  id integer auto_increment PRIMARY KEY,  
  title varchar(100),  
  abstract text);
```

Após a criação da tabela, cria-se a aplicação “teste_ror” com o seguinte comando:

```
rails teste_ror
```

A saída resultante da execução deste comando é mostrada na Figura 3.10.



```
Arquivo  Editar  Ver  Terminal  Abas  Ajuda  
  
create  script/performance  
create  script/process  
create  test/fixtures  
create  test/functional  
create  test/integration  
create  test/mocks/development  
create  test/mocks/test  
create  test/unit  
create  vendor  
create  vendor/plugins  
create  tmp/sessions  
create  tmp/sockets  
create  tmp/cache  
create  tmp/pids  
create  Rakefile  
create  README  
create  app/controllers/application.rb  
create  app/helpers/application_helper.rb  
create  test/test_helper.rb  
create  config/database.yml  
create  config/routes.rb  
create  public/.htaccess  
create  config/boot.rb  
create  config/environment.rb
```

Figura 3.10: Saída resultante do comando *rails*

Neste momento, pode-se configurar o arquivo **config/database.yml**, que configura corretamente a instância de banco de dados que é utilizada. Após realizada esta configuração, pode ser gerado o primeiro código da aplicação, através do gerador de *scaffold*. Finalmente, o seguinte comando pode ser executado a partir do diretório raiz da aplicação (teste_ror):

```
script/generate scaffold paper
```

A saída deste comando é a seguinte:

```
exists  app/controllers/  
exists  app/helpers/  
create  app/views/papers  
exists  app/views/layouts/  
exists  test/functional/  
dependency model  
exists  app/models/  
exists  test/unit/  
exists  test/fixtures/  
create  app/models/paper.rb  
create  test/unit/paper_test.rb
```

```
create test/fixtures/papers.yml
create app/views/papers/_form.rhtml
create app/views/papers/list.rhtml
create app/views/papers/show.rhtml
create app/views/papers/new.rhtml
create app/views/papers/edit.rhtml
create app/controllers/papers_controller.rb
create test/functional/papers_controller_test.rb
create app/helpers/papers_helper.rb
create app/views/layouts/papers.rhtml
create public/stylesheets/scaffold.css
```

Após a criação do *scaffold*, pode-se executar o servidor WebRick, que é distribuído juntamente com o **Rails** e é bastante útil para testar as aplicações sendo desenvolvidas. O seguinte comando a partir do diretório da aplicação é suficiente para executar o servidor:

```
script/server
```

Com este comando, o servidor passa a esperar conexões HTTP na porta 3000 (padrão do WebRick com **Rails**). Portanto, apontando o *browser* para o endereço `http://localhost:3000/papers`, acessa-se a página exibida na Figura 3.11. Esta tela é feita para listar todos os registros que forem incluídos no banco. Além desta tela, através do gerador *scaffold* são disponibilizadas telas para inserção de registros (Figura 3.12), exibição de registros (Figura 3.13) e edição de registros (Figura 3.14).

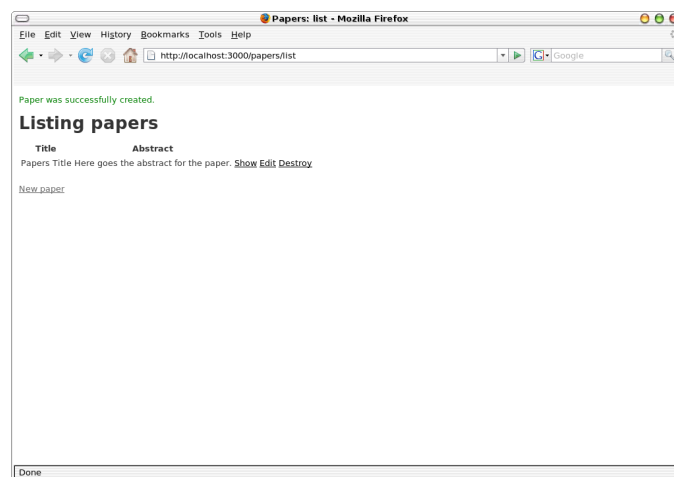


Figura 3.11: Tela para listagem de papers

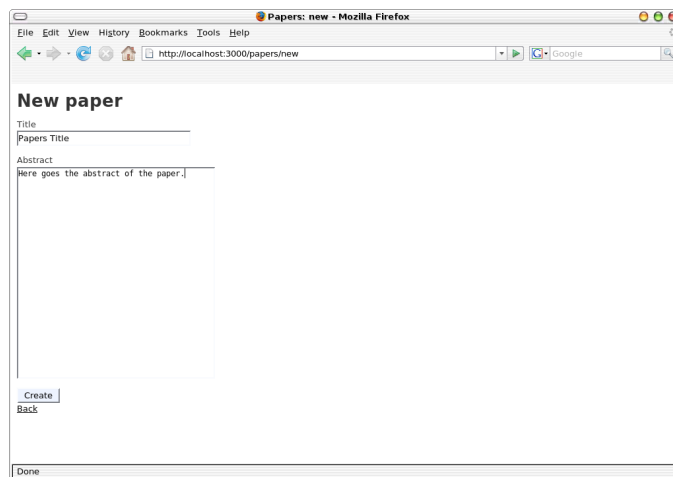


Figura 3.12: Tela para inclusão de papers

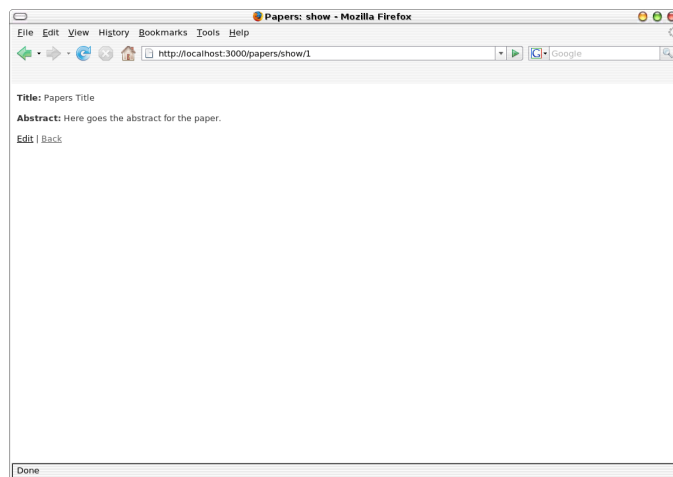


Figura 3.13: Tela para exibição de papers

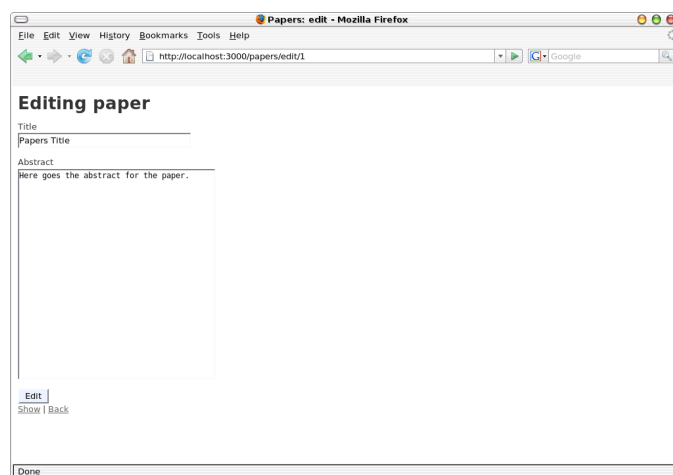


Figura 3.14: Tela para edição de papers

Observando a saída resultante do comando utilizado para criar o *scaffold*, é possível observar que foi criada uma aplicação em **Ruby on Rails** com os arquivos corretamente distribuídos pelas camadas de modelo, visão e controle. Cada uma destas camadas, disponibilizada pelo *framework*, é detalhada nas seções a seguir.

3.3.6. Modelo: o *framework* ActiveRecord

O *ActiveRecord* é o *framework* do **Ruby on Rails** que conecta os objetos de código (as instâncias do modelo conceitual da Camada de Negócios) com as tabelas do banco de dados para criar um modelo do domínio persistente, no qual as lógicas e os dados são apresentados conjuntamente. Ele é uma implementação do padrão mapeamento objeto-relacional (ORM – *Object-Relational Mapping*) chamado *Active Record*.

A principal contribuição do padrão de projeto ORM ou *Active Record* é auxiliar em dois problemas: falta de associações e herança. Ao adicionar uma linguagem de macros do domínio simples para descrever as associações e integrar o padrão *Single Table Inheritance* para descrever a herança, o *Active Record* reduz o *gap* de funcionalidades entre o mapeador de dados e a abordagem de registro ativo. Suas principais funcionalidades são:

- **mapeamento automatizado entre as classes e tabelas, atributos e colunas:** no exemplo desenvolvido na Seção 3.3.5 fica claro o mapeamento automático. No caso, não foi feita qualquer configuração, mas o arquivo **app/model/paper.rb** foi criado pelo *scaffold*. O conteúdo deste arquivo é exibido na Figura 3.15

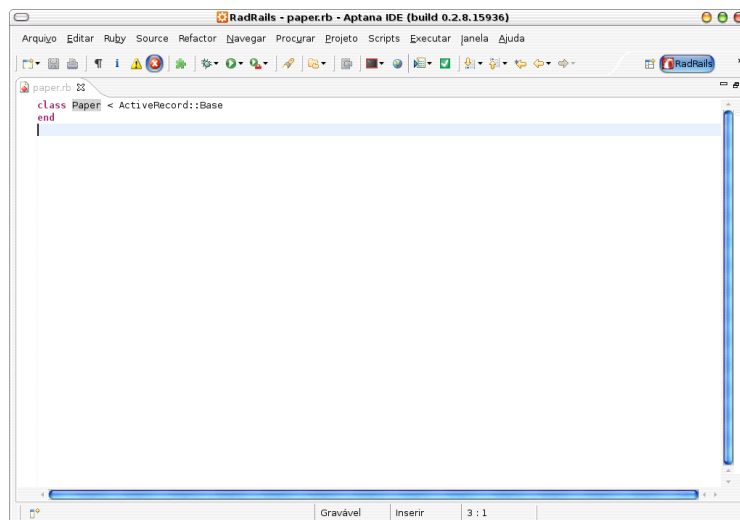


Figura 3.15: Classe referente ao Modelo conceitual de Paper

Como é possível observar na Figura 3.15, a classe do modelo está vazia. Embora ela esteja vazia, seu mapeamento com a tabela *papers* é feita automaticamente, utilizando-se as regras de pluralização – este é um exemplo clássico da aplicação do conceito “*Convention over Configuration*”. Não é necessária qualquer configuração para que o modelo funcione corretamente, com os métodos *read* e *save* implementados (herdados).

É importante notar que as classes do modelo devem obrigatoriamente herdar a classe “*ActiveRecord::Base*”. Além de ter sido feito o mapeamento da classe *Paper* na tabela *papers*, os campos da tabela podem ser acessados como atributos da classe,

portanto, se “paper1” for uma instância de “Paper”, poderemos acessar os atributos “title” e “author”; as operações de leitura e escrita dos registros no banco são feitas pelos métodos “read”, “write”, “find”, entre outros.

- **associações entre os objetos controlados através de macros simples de meta-programação:** caso o nosso modelo contemplasse, por exemplo, as entidades Paper e Author, com um relacionamento de muitos-para-muitos entre as entidades, poderíamos declarar esta associação da seguinte forma:

```
class Paper < ActiveRecord::Base
  has_many_and_belongs_to_many :author
end
```

De maneira análoga, podemos declarar outros tipos de associação: *has_many*, *belongs_to*, *has_one*.

- **agregação de objetos de valor controlados por macros simples de meta-programação:** caso existam agregação, podem ser utilizadas as macros “composed_of”. No exemplo do artigo, não são utilizadas agregações, mas há vários casos em que podem ser necessárias, conforme o exemplo que segue:

```
class Account < ActiveRecord::Base
  composed_of :balance, :class_name => "Money",
    :mapping => %w(balance amount)
  composed_of :address,
    :mapping => [%w(address_street street), %w(address_city city)]
end
```

- **regras de validação (que podem ser diferentes para objetos novos e para os já existentes):** existem várias facilidades para se fazer a validação dos dados no servidor. Os validadores geram mensagens para o usuário no caso em que as regras não são satisfeitas. Como exemplo, no caso da inclusão de papers pode-se incluir um registro com o “title” em branco. Utilizando as regras de validação, é possível bloquear este tipo de comportamento:

```
class Paper < ActiveRecord::Base
  validates_presence_of :title, :message=>'must be present for every paper'
  validates_presence_of :abstract
  validates_length_of :title, :minimum=>3
end
```

Com a classe Paper alterada conforme o código acima, caso tente-se inserir um paper com o title e abstract em branco, a tela da Figura 3.16 é exibida para o usuário.

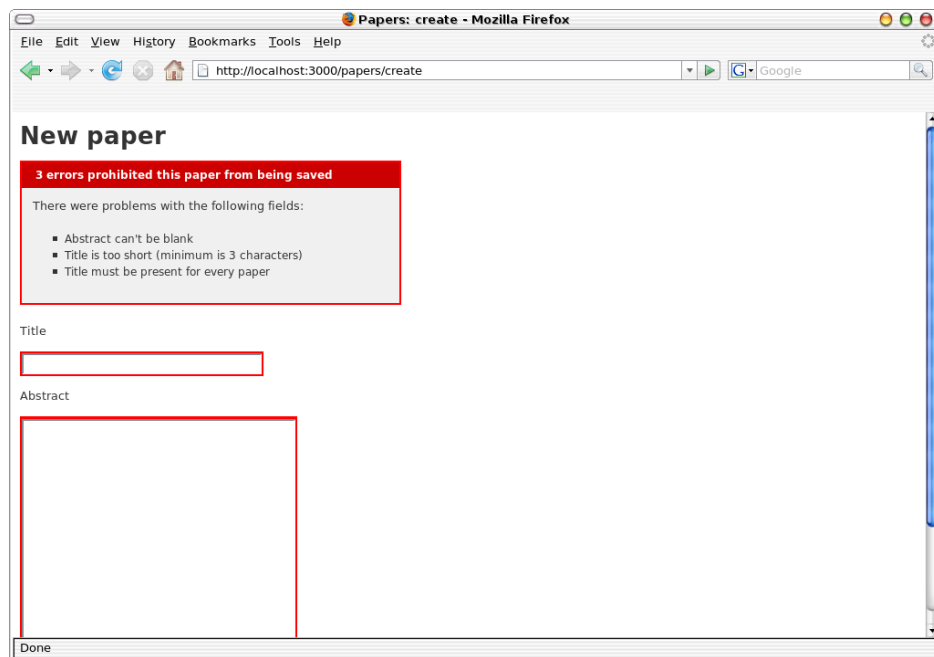


Figura 3.16: Exemplo da utilização de validação

- ações podem fazer os registros funcionarem como listas ou árvores:

```
class Paper < ActiveRecord::Base
  has_many :keywords, :order => "position"
end
class Keyword < ActiveRecord::Base
  belongs_to :paper
  act_as_list :scope => :paper
end

paper.first.move_to_bottom
paper.last.move_higher
```

- **callbacks como métodos ou filas em todo o ciclo de vida (criar, salvar, destruir, validar, etc.):** *callbacks* permitem que sejam disparadas ações (lógica) antes ou depois da alteração de estado de um objeto. Exemplos são: *before_save* e *before_create*.
- **“Observers” durante todo o ciclo de vida:** “observers” são classes que respondem aos *callbacks* para implementar comportamento semelhante ao de *triggers* fora da classe original. Por exemplo, para o caso de registro de artigos, podemos implementar um observer que envia um email quando determinado artigo é armazenado no banco de dados:

```
class PaperObserver < ActiveRecord::Observer
  def after_save(paper)
    Notifications.deliver_comment("admin@paperssite.com", "Novo paper adicionado",
    paper)
  end
end
```

- **hierarquias por herança:**

```
class Paper < ActiveRecord::Base; end
class ShortPaper < Paper; end
```

- **suporte a transações:** são permitidas “transactions” tanto nos objetos como no nível do banco de dados.
- **reflexão em colunas, agregações e associações:** o mecanismo de reflexão permite que as classes e objetos do Active Record sejam questionadas sobre suas associações e agregações
- **manipulação direta** (ao invés de invocação de serviço): ao invés de fazer (exemplo do Hibernate):

```
long pkId = 1234;
DomesticCat pk = (DomesticCat) sess.load( Cat.class, new Long(pkId) );
// algo interessante envolvendo um gato
sess.save(cat);
sess.flush(); // force the SQL INSERT
```

O *Active Record* permite que se faça o mesmo, de maneira mais simplificada, como a seguir:

```
pkId = 1234
cat = Cat.find(pkId)
# something even more interesting involving the same cat...
cat.save
```

- **abstração de banco de dados através da utilização de adaptadores simples com conector compartilhado:** o *Active Record* pode se conectar com praticamente qualquer banco de dados, pois utiliza abstração do banco de dados. Segundo a documentação do *framework*, é possível criar um adaptador para praticamente qualquer banco com aproximadamente 100 linhas de código. Ele possui suporte embutido para MySQL, PostgreSQL, SQLite, Oracle, SQLServer e DB2.
- **logging de Log4r e Logger:** os mecanismos para *logging* são semelhantes aos que são disponibilizados em outras linguagens e *frameworks* para desenvolvimento de aplicações voltadas para a *web*

Nesta seção, foi dada uma visão geral do *framework ActiveRecord* e suas principais funcionalidades. Algumas funcionalidades foram mostradas no exemplo da

aplicação exemplo desenvolvida na Seção 3.3.5, e outras funcionalidades não foram exploradas em detalhes. Entretanto, pode-se perceber que o *framework* disponibiliza recursos suficientes para o desenvolvimento de um grande número de aplicações *web* que utilizam recursos de bancos de dados.

Por se tratar de um capítulo introdutório sobre **Ruby on Rails**, diversos aspectos do *framework* não foram tratados, assim como questões relacionadas a melhoria de *performance*, escalabilidade, entre outras.

3.3.7. Controle: *ActionController*

Os arquivos que compõem a camada de controle em **Ruby on Rails** ficam armazenados no diretório **app/controller** da aplicação. Esta é a camada responsável por tratar as requisições HTTP dos usuários. Os controles são feitos de uma ou mais ações que são executadas quando as requisições HTTP ocorrem; cada ação pode tanto renderizar um *template* quanto redirecionar para outra ação. Uma ação é definida como um método público no controlador, que será acessível pelo servidor web através dos roteamentos do **Ruby on Rails**.

No exemplo desenvolvido na Seção 3.3.5, o controlador é chamado **papers_controller.rb**, e seu conteúdo é igual ao exibido abaixo:

```
class PapersController < ApplicationController
  def index
    list
    render :action => 'list'
  end
  # GETs should be safe (see
  # http://www.w3.org/2001/tag/doc/whenToUseGet.html)
  verify :method => :post, :only => [ :destroy, :create, :update ],
        :redirect_to => { :action => :list }
  def list
    @paper_pages, @papers = paginate :papers, :per_page => 10
  end
  def show
    @paper = Paper.find(params[:id])
  end
  def new
    @paper = Paper.new
  end
  def create
    @paper = Paper.new(params[:paper])
    if @paper.save
      flash[:notice] = 'Paper was successfully created.'
      redirect_to :action => 'list'
    else
      render :action => 'new'
    end
  end
end
```

```

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

```

Como é possível observar, existem diversas ações que foram criadas pelo gerador *scaffold*: *index*, *list*, *show*, *new*, *create*, *edit*, *update*, *destroy*. Além disso, que as classes dos controladores devem **sempre** ser uma especialização da classe ApplicationController. Esta classe ApplicationController é criada automaticamente pelo rails quando a aplicação é criada com o comando *rails*. Ela é uma especialização da classe ActionController::Base.

Outro detalhe que deve ser notado na classe PapersController é que no início é feita uma verificação para algumas ações, que só devem receber requisições do tipo HTTP POST: as ações *destroy*, *create* e *update*. Estas ações não possuem um template correspondente na camada de visão, como é detalhado na Seção 3.3.8.

Para as demais ações, é aplicada a seguinte regra: primeiramente o método correspondente à ação na classe de controle é executado, as classes do modelo que forem alteradas ficam disponíveis para a aplicação com as informações atualizadas e, então, um *template* é renderizado na camada de visão. Este *template* terá acesso aos objetos do modelo que foram manipulados na camada de controle.

O *ActionController*, no **Ruby on Rails**, é implementado no *framework* *ActionPack*, que também contém a *ActionView*. **Ruby on Rails** fornece mecanismos para armazenamento de objetos em sessão, armazenamento de sessões no sistema de arquivos ou em banco de dados, entre outros. Entretanto, neste capítulo não são detalhadas as outras características da camada de controle.

3.3.8. Visão: *ActionView*

A camada de visão, como o nome diz, contém o conteúdo que é disponibilizado para os usuários da aplicação. É através dela que os usuários interagem com a aplicação, realizando as ações desejadas e visualizando os dados disponibilizados pela aplicação.

Os arquivos desta camada são disponibilizados no diretório **app/views**. Existe um diretório que é criado por padrão para todas as aplicações chamado **layouts**. Neste diretório são armazenados os *layouts* da aplicação, que são arquivos escritos na linguagem *eRuby*, uma linguagem HTML com inclusão de sintaxe Ruby. Para utilizar **Ruby** nos arquivos, devem ser utilizadas as *tags* `<%` e `%>` ou `<%= %>` quando o resultado deve ser mostrado no HTML resultante.

Na aplicação de exemplo mostrada na Seção 3.3.5 são criados automaticamente pelo gerador *scaffold* os arquivos **layouts/papers.rhtml**, que contém o layout geral de todas as páginas relacionadas a papers, e **papers/edit.rhtml**, **papers/_form.rhtml**, **papers/list.rhtml**, **papers/new.rhtml**, **papers/show.rhtml**.

O arquivo **layouts/papers.rhtml** é utilizado para todas as páginas relacionadas à entidade paper. Seu conteúdo é listado abaixo:

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
4 <head>
5   <meta http-equiv="content-type" content="text/html; charset=UTF-8" />
6   <title>Papers: <%= controller.action_name %></title>
7   <%= stylesheet_link_tag 'scaffold' %>
8 </head>
9 <body>
10  <p style="color: green"><%= flash[:notice] %></p>
11  <%= yield %>
12 </body>
13 </html>
```

Neste arquivo, é importante destacar o conteúdo da linha 11: ela invoca o comando *yield* e disponibiliza o conteúdo. Com isso, é incluído o bloco do *template* de cada *view* conforme as ações são realizadas. Portanto, é nesta parte que cada *template* disponível em **views/papers/** será inserido.

Como exemplo, segue abaixo o código do arquivo **views/papers/list.rhtml**, que foi gerado automaticamente pelo gerador *scaffold*:

```

1 <h1>Listing papers</h1>
2 <table>
3 <tr>
4 <% for column in Paper.content_columns %>
5 <th><%= column.human_name %></th>
6 <% end %>
7 </tr>
8 <% for paper in @papers %>
9 <tr>
10 <% for column in Paper.content_columns %>
11 <td><%= h paper.send(column.name) %></td>
12 <% end %>
13 <td><%= link_to 'Show', :action => 'show', :id => paper %></td>
14 <td><%= link_to 'Edit', :action => 'edit', :id => paper %></td>
15 <td><%= link_to 'Destroy', { :action => 'destroy', :id => paper },
16 :confirm => 'Are you sure?', :method => :post %></td>
17 </tr>
18 <% end %>
19 </table>

```

O resultado deste código é a tela que é exibida na Figura 3.11. Para a geração da tela, os seguintes eventos ocorreram:

- Usuário acessou o endereço `http://localhost:3000/papers`
- *Controller* foi disparado na ação *index*, que redireciona para *list*
- O método *list* da classe *PapersController* foi executado, inicializando as variáveis `@paper_pages` e `@papers`
- O arquivo **`views/layouts/papers.rhtml`** foi processado, e, quando o comando *yield* foi invocado ele inclui o arquivo **`views/papers/list.rhtml`**
- Os objetos do modelo foram todos disponibilizados para a *view*, que exibiu seus resultados na tela

O método “`link_to`” é utilizado no arquivo **`list.rhtml`**. Este método cria links na *view*, que permitem a interação do usuário. Este método, implementado na classe `ActionView::Helpers::UrlHelper`, recebe os seguintes argumentos: nome, opções como uma hash, opções de html, e parâmetros para referência do método utilizado.

Este método gera o código HTML automaticamente, inclusive o código JavaScript que for necessário para executar determinadas ações, como no exemplo, em que deve ser feita a confirmação da exclusão de determinado registro. Cada clique em link será tratado pela camada de controle, conforme a *action* indicada. Além da indicação da ação, este método também aceita como parâmetro no hash de opções a indicação do controlador que irá tratar esta ação, ou seja, qual a classe da camada de controle que contém a ação a ser executada.

Outra forma de disparar eventos na camada de visão é através da criação de forms. **Ruby on Rails** também possui métodos que auxiliam na criação de formulários, como pode ser visto no código do arquivo **app/views/papers/new.rhtml** e **app/views/papers/_form.rhtml**, que renderizam a tela mostrada na Figura 3.12. Abaixo está o código do arquivo **app/views/papers/new.rhtml**:

```
1 <h1>New paper</h1>
2 <% form_tag :action => 'create' do %>
3   <%= render :partial => 'form' %>
4   <%= submit_tag "Create" %>
5 <% end %>
6 <%= link_to 'Back', :action => 'list' %>
```

No código listado acima, pode-se observar que na linha 2 é iniciado um formulário HTML com o método **form_tag**. Este método cria automaticamente as tags para criação de formulário html, configurando a *action* corretamente. Na linha 3, é invocado o método “render”, e, como parâmetro, é passado o argumento “:partial => form”. Este conceito é o de renderização de partes de tela. Um arquivo que contém uma parte pode ser reutilizado, o que é bastante comum no caso de forms. Por exemplo, tanto a ação de criação de paper como a edição utilizam o mesmo form, embora de maneiras um pouco diferentes. Portanto, neste caso, é feita a renderização do arquivo **_form.rhtml**, cujo código é listado abaixo:

```
1 <%= error_messages_for 'paper' %>
2 <!--[form:paper]-->
3 <p><label for="paper_title">Title</label><br/>
4 <%= text_field 'paper', 'title' %></p>
5 <p><label for="paper_abstract">Abstract</label><br/>
6 <%= text_area 'paper', 'abstract' %></p>
7 <!--[eoform:paper]-->
```

As linhas 2 e 7 servem apenas como comentário, elas não têm qualquer efeito sobre o processamento dos arquivos de *templates*. Nas linhas 3 e 5, são criados rótulos para os elementos do formulário. Na linha 4 é criado um campo de texto com a utilização do método **text_field**, enquanto na linha 6 é criado uma área de texto com o método **text_area**.

O elemento “submit”, que cria um botão no momento da renderização, é criado diretamente no arquivo **app/views/papers/new.rhtml** (linha 4), pois o texto dele é diferente nas telas de criação e edição de arquivos.

Nesta seção, apresentamos os principais detalhes da camada de visão das aplicações **Ruby on Rails**. Entretanto, muitos *helpers* já foram desenvolvidos para o *framework*, sendo possível gerar telas de login de forma automática, páginas com Ajax, e diversos outros tipos de interface geralmente necessários. Os principais projetos desenvolvidos em **Ruby on Rails** podem ser acessados em <http://www.rubyforge.org>.

Neste endereço, é possível obter geradores de diversas partes das aplicações desenvolvidas em *rails*.

3.3.9. Testes de unidade e funcionais

Ruby on Rails disponibiliza mecanismos para a realização de testes de unidade e funcionais para todas as aplicações. Os testes são realizados no banco de dados de teste, que tem seus dados excluídos e reconfigurados a cada vez que os testes são executados.

As classes com os testes estão localizadas no diretório **test/**. Em **test/unit**, estão os testes de unidade. É criada uma classe de teste para cada elemento do modelo. No caso da aplicação criada como exemplo na Seção 3.3.5, foram criados os arquivos **unit/paper_test.rb** e **functional/papers_controller_test.rb**. Para a realização dos testes, devem ser editados os arquivos de teste de unidade e de integração, inserindo-se as assertivas desejadas.

3.3.10. Distribuição de aplicações em Ruby on Rails

As aplicações desenvolvidas em **Ruby on Rails** possuem um *Rakefile* que é criado automaticamente pela ferramenta **rails** durante a criação do projeto. Para distribuir uma aplicação, pode-se configurar o ambiente de produção, ou documentar sua configuração, e distribuir os arquivos da aplicação. A ferramenta *rake* é então utilizada para processar o *Rakefile*, e pode construir o ambiente de produção com o mínimo de intervenção possível. É necessário que se configure a base de dados de produção.

Entretanto, quando uma aplicação for distribuída, ela dependerá do *framework rails*, que deve estar instalado no sistema de um cliente, no qual a aplicação será instalada. O cliente possui liberdade para alterar os arquivos do *framework*, seja corrompendo-os, atualizando-os, ou fazendo qualquer outra operação. Para evitar a dependência do ambiente do cliente, e garantir que a configuração que foi distribuída é a que continuará sendo utilizada pelo cliente, foi criado o mecanismo de *freezing* das aplicações desenvolvidas com o *framework Ruby on Rails*.

O mecanismo de *freezing* – ou congelamento – consiste em copiar todos os arquivos do *framework* para o diretório **vendor** da aplicação que será distribuída. Para fazer isso, o seguinte comando deve ser executado a partir do diretório raiz da aplicação:

```
rake rails:freeze:gems
```

3.4. Ambiente de desenvolvimento

Nas Seções 3.2 e 3.3 foram apresentados, respectivamente, os principais conceitos da linguagem **Ruby** e do *framework* **Ruby on Rails**. As informações apresentadas utilizaram exemplos simples para ilustrar os principais conceitos do *framework*, que podem ser então aplicados na criação de aplicações de maior porte.

Para a configuração de um ambiente de desenvolvimento de aplicações maiores com **Ruby on Rails**, é sugerido um conjunto de ferramentas de apoio. Estas ferramentas têm o objetivo de organizar um ambiente de desenvolvimento, que pode ter vários desenvolvedores trabalhando paralelamente num mesmo projeto [Thomas and Hansson 2006]. As ferramentas sugeridas para o desenvolvimento das aplicações são:

- **Bazaar** (<http://bazaar-vcs.org>) – esta é uma ferramenta de controle de versões com repositórios distribuídos. Ela é utilizada para o controle de versões do código-fonte da distribuição Linux Ubuntu, e assemelha-se à ferramenta git, utilizada para o controle da evolução do kernel do sistema operacional Linux.
- **Bugzilla** (<http://www.bugzilla.org>) – a configuração e utilização do bugzilla não será detalhada neste material, mas é recomendada a instalação da ferramenta para a execução das atividades de rastreamento de bugs nas aplicações desenvolvidas e também de controle de mudanças. Bugzilla é muitas vezes utilizada em substituição aos sistemas de Service Desk das empresas.
- **RadRails** (<http://www.radrails.org>) – Esta é uma IDE de código aberto para o desenvolvimento de aplicações com Ruby on Rails. Seu uso é recomendado por facilitar o desenvolvimento de aplicações com o *framework*. Além desta IDE, a equipe do Netbeans (<http://www.netbeans.org>) está se esforçando para colocar todo o suporte a **Ruby on Rails** na ferramenta.

O ambiente de desenvolvimento sugerido aqui estimula a separação de papéis no desenvolvimento. Os “papéis” que estão sendo representados aqui são:

- **Programador da aplicação:** responsável por fazer a programação da aplicação sendo desenvolvida;
- **Designer da aplicação:** responsável por organizar o *layout* da aplicação sendo desenvolvida. Na medida do possível, deve sofrer a menor interferência possível do programador;
- **Administrador de banco de dados:** responsável por fazer as alterações em banco de dados (criação do banco, etc). Em ambiente de produção, também é o responsável por tratar as questões de *performance* e escalabilidade do banco.

3.4.1 Instalação do Ruby on Rails

O *framework* Ruby on Rails está disponível no endereço eletrônico <http://www.rubyonrails.org>. Ele é distribuído em formatos binários e de código fonte, e os binários estão incluídos em muitas das distribuições Linux, ou em repositórios de código.

Para instalar o **Ruby on Rails** na distribuição Linux Ubuntu 7.04, basta executar o seguinte comando quando a máquina estiver conectada à internet:

```
> sudo apt-get install rails
```

Após feita a instalação do pacote, a primeira aplicação pode ser criada com o comando:

```
> rails [nome_da_aplicacao]
```

Como exemplo, para criarmos a aplicação chamada “app1”, já é possível executar um servidor HTTP simples que é disponibilizado junto com o Ruby on Rails da seguinte forma:

```
> rails app1  
> cd app1  
> script/server
```

Com isso, o servidor ficará rodando automaticamente na porta 3000, e podemos validar a criação de nossa nova aplicação e seu respectivo servidor acessando, através de um navegador web, o endereço <http://localhost:3000>. Na Figura 3.17 é mostrada a tela que deve ser exibida.

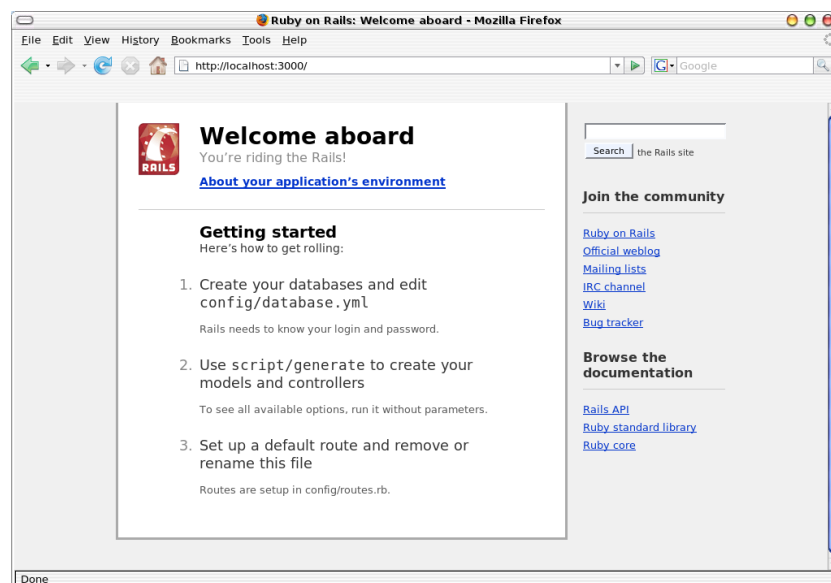


Figura 3.17: Tela inicial de uma aplicação configurada corretamente

Além da opção de se criar manualmente uma aplicação, podemos utilizar também uma IDE (*Integrated Development Environment*) para desenvolvimento de

aplicações com **Ruby on Rails**. A IDE *RadRails*, baseada na *Aptana*, auxilia no trabalho de desenvolvimento. Na Figura 3.18 é exibida a interface principal da IDE.

3.4.2. Instalação da IDE Aptana

A IDE **Aptana** pode ser obtida no endereço <http://www.apтана.com>. Como ela é baseada na tecnologia Java, o computador na qual ela será instalada deve ter a última versão da Java Virtual Machine da Sun, que pode ser obtida no endereço <http://java.sun.com>.

Para instalação da IDE, tanto em ambiente Windows quanto Linux, pode ser feito *download* da versão compactada em formato zip da IDE, e então a descompactação no computador do usuário. A partir deste momento, a IDE está pronta para ser iniciada. A partir da IDE, é possível estabelecer conexão com os bancos de dados configurados para a aplicação dos projetos desenvolvidos em **Rails**, utilizar os geradores de código, montar os pacotes para distribuição das aplicações, entre outros.

A tela principal da IDE é exibida na Figura 3.18.

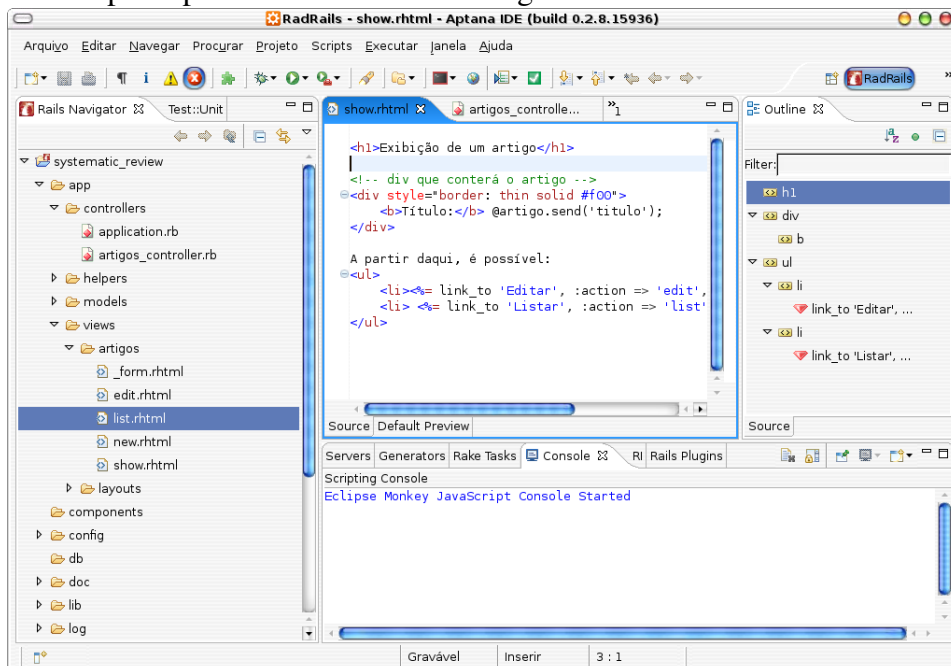


Figura 3.18: Tela principal da IDE RadRails

3.4.3. Bazaar – configuração de repositórios distribuídos

Durante o desenvolvimento de sistemas computacionais, uma grande quantidade de informações é gerada: arquivos de código-fonte, documentação, scripts de banco de dados, arquivos de configuração, entre outros. De alguma maneira, deve ser feito um controle sobre a evolução destes itens durante o processo de desenvolvimento, em todas as suas fases, desde o início do projeto até a fase de manutenção e, eventualmente, descontinuidade de um projeto de software.

Historicamente, grande parte dos projetos de software utiliza repositórios centralizados para o controle de versões de seus arquivos. Como exemplo, sabe-se que o CVS³ foi por muito tempo uma das ferramentas mais utilizadas para a realização do controle de versões de código-fonte, e, ainda hoje, é uma das ferramentas mais utilizadas nos projetos de software livre [Reis e Fortes, 2002]. Muitas vezes em que ela não é mais utilizada, foi substituída por Subversion, que possui os mesmos conceitos de repositórios centralizados e trabalho baseado no modelo *checkout-commit*. Este modelo de controle de versões define que cada desenvolvedor possui uma cópia local, também chamada de “cópia de trabalho”, na qual o desenvolvedor pode aplicar suas alterações. Assim que finalizar uma alteração, o desenvolvedor a envia para o repositório central, através da operação de *commit*. Algumas iniciativas foram feitas para melhorar a utilização do sistema, melhorando a usabilidade, criando clientes voltados para o ambiente *web* e permitindo a configuração de acesso em repositórios CVS [Junqueira e Fortes, 2004].

Embora seja um modelo bastante utilizado e permita um controle efetivo da evolução dos arquivos que compõem determinado sistema, este modelo pode não funcionar corretamente em equipes muito grandes e geograficamente dispersas – o que tem se tornado cada vez mais comuns nos projetos de software. Algumas das características que poderiam ser melhoradas são:

- **área de trabalho local:** neste modelo, cada desenvolvedor possui apenas a última versão da árvore (*branch*) em que está trabalhando, ou alguma outra versão recuperada do repositórios
- **repositório centralizado:** existe apenas um repositório para todos os desenvolvedores envolvidos no projeto. O sucesso do projeto depende da boa utilização do repositório por todos os desenvolvedores, pois o trabalho de cada um pode ser diretamente afetado pelo trabalho de todos os outros, e um desenvolvedor mal intencionado poderia ocasionar bastante transtorno

Estas características, embora possam ser apropriadas para muitos projetos de software, podem também atrapalhar o desenvolvimento de muitos outros projetos, e certamente existe o problema de se ter uma forte dependência das políticas de utilização do repositório – neste modelo, muito esforço pode ser despendido para a resolução de conflitos que ocorrem no desenvolvimento concorrente.

Para resolver os problemas que podem surgir com este modelo, foram desenvolvidos sistemas de controle de versões que permitem o trabalho com repositórios distribuídos, sendo o *BitKeeper*, da empresa *BitMover*, um dos primeiros sistemas de controle de versões com repositórios distribuídos de maior destaque. Ele foi utilizado

3 <http://savannah.nongnu.org/projects/cvs/>

entre os anos de 2002 e 2005 para o controle de versões do *kernel* do Linux, sendo então substituído pelo *git*⁴, um sistema de código aberto desenvolvido pela comunidade de software livre, que foi arquitetado por Linus Torvalds. Além do *git*, outro sistema de código aberto que utiliza este conceito é o *bazaar*, desenvolvido e mantido pela comunidade de software livre e pela Canonical Inc. - este também é o sistema de controle de versões utilizado para controlar o desenvolvimento da distribuição *Ubuntu*.

O modelo de repositórios distribuídos possui as seguintes características que os diferenciam dos sistemas com repositório centralizado:

- **repositório distribuído:** não existe um repositório centralizado com todo o código fonte, embora alguns repositórios possam ser criados para centralizar o desenvolvimento do sistema. Cada desenvolvedor, ao invés de trabalhar numa única versão do repositório, possui uma cópia integral de um repositório, com todas as versões armazenadas, incluindo os diversos *branches*, caso existam
- **sincronização entre repositórios:** ao invés de ocorrer sincronização entre uma única versão do desenvolvedor com o repositório central, os sistemas com repositórios distribuídos fazem a sincronização entre repositórios, realizando a comparação de todas as versões modificadas
- **foco em integração:** enquanto o foco dos sistemas com repositório centralizado está na “separação” do código entre os desenvolvedores, os sistemas com repositório distribuído focam principalmente na “integração” do código, ou seja, nas operações de junção (*merge*)

No ambiente de trabalho com repositórios distribuídos, cada desenvolvedor trabalha sobre um repositório completo, e não apenas em uma versão dos arquivos – o que permite, inclusive, que o desenvolvedor trabalhe de forma *off-line*. As operações de *checkout* e *commit* realizadas pelos desenvolvedores não afetam o trabalho de quaisquer outros desenvolvedores, ocorrendo apenas em seus repositórios locais. Além disso, cada repositório pode ser sincronizado com todos os outros; existem duas operações básicas para o trabalho com repositórios distribuídos: *push* e *pull*. A operação de *push* consiste em um desenvolvedor enviar suas alterações para outro, postando-as no repositório de outro desenvolvedor. A operação de *pull* é o inverso: permite a um desenvolvedor recuperar as alterações de outro para seu próprio repositório, fazendo o *merge*.

4 <http://git.or.cz/>

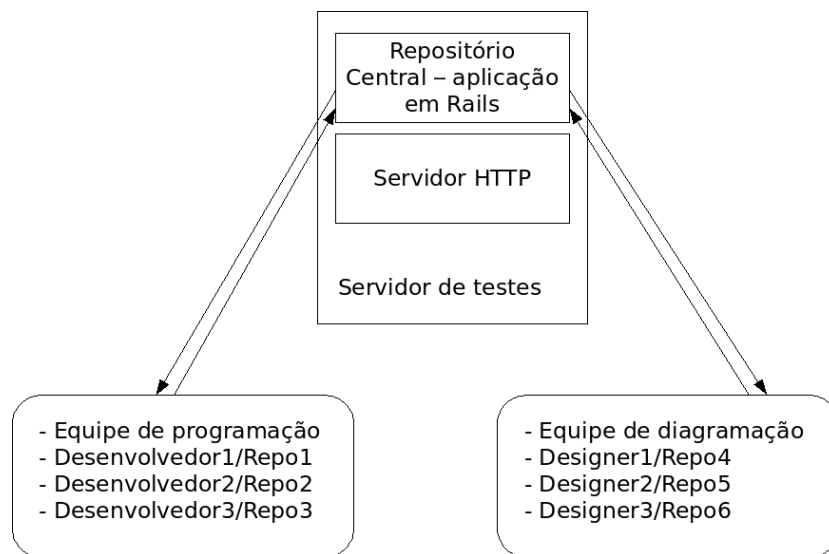


Figura 3.20: Representação do esquema de trabalho com repositórios distribuídos

Para o desenvolvimento de aplicações *web* com **Ruby on Rails**, é possível estabelecer um fluxo de trabalho para o desenvolvimento, com atualização automática dos resultados exibidos em servidores com acesso compartilhado, de forma que cada desenvolvedor pode realizar seu trabalho em seu repositório local, e, quando for conveniente, sincroniza suas alterações com o repositório remoto, publicando então suas alterações para outros desenvolvedores.

A abordagem para o controle de versões proposta nesta seção considera que haverá um servidor remoto, configurado para permitir sincronização a partir de todos os outros repositórios. Então, cada equipe de trabalho pode compartilhar seus repositórios entre si, e, a cada etapa concluída e testada, pode publicar suas alterações para o repositório centralizado, de onde as outras equipes de trabalho podem recuperar os dados. Como exemplo, na Figura 3.20 é representado o esquema em que cada desenvolvedor e cada *designer* possui seu próprio repositório (**RepoN**), e os membros das equipes de programação e *design* podem compartilhar os repositórios, fazendo *merge* entre eles, mas não pode ser feito *merge* diretamente entre as equipes.

A vantagem de se trabalhar em repositórios distribuídos com a separação de papéis é que pode ser diminuída a interferência nos trabalhos das equipes, sendo que cada uma tem um “acordo” com as demais equipes de manter o sistema funcionando no servidor de testes. Cada programador e cada *designer* pode realizar todos os testes localmente antes de enviar para o servidor centralizado. Desta forma, cada equipe e cada pessoa envolvida no desenvolvimento de um aplicação pode focar seus esforços nas atividades que estiver realizando.

3.5. Perspectivas do Desenvolvimento Web

Por muito tempo, os sistemas computacionais desenvolvidos eram compostos, basicamente, por aplicações *stand-alone* que rodavam em *Desktops* ou aplicações com a arquitetura *cliente-servidor* que possuíam uma interface gráfica capaz de rodar como *Desktop*.

No entanto, no início da década de 1990, foi introduzida a World Wide Web com o objetivo de deixar as informações disponíveis de uma forma simples e consistente. Por muitos anos, a *web* evoluiu e se popularizou, mas seu protocolo básico de comunicação manteve-se praticamente inalterado. Até o fim da década de 1990, observou-se um crescimento significativo da Internet, e foi revelado seu potencial como meio para realização de negócios.

As páginas da World Wide Web passaram a evoluir de um formato estático, formado por páginas HTML relativamente simples, embora com grandes evoluções na parte visual, para um formato dinâmico, em que não são apenas disponibilizadas as informações, mas são construídas aplicações que são verdadeiras plataformas sobre as quais são realizados negócios no mundo todo.

Para o desenvolvimento destas aplicações, que situam-se num contexto diferente das aplicações desenvolvidas para *Desktop*, foram criados *frameworks* com o objetivo de auxiliar o trabalho de codificação e manutenção das aplicações, que devem lidar geralmente com um público heterogêneo e com grandes quantidades de dados.

Segundo [Jazayeri, 2007], a maioria destes *frameworks* baseia-se atualmente em técnicas de Orientação a Objetos e no padrão MVC, que permite a divisão da aplicação em camadas: Modelo, Visão e Controle. **Ruby on Rails** se apresenta como boa opção para o desenvolvimento de aplicações *web* por facilitar a organização das aplicações no padrão MVC, além de ser uma linguagem de tipagem dinâmica, o que colabora para a redução de erros de desenvolvimento. **Ruby on Rails** também facilita bastante o trabalho com grandes quantidades de dados, e fornece mecanismos para auxiliar na manipulação destes dados.

Referências

- Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. 1999 *Hypertext Transfer Protocol -- Http/1.1*. RFC. RFC Editor.
- Jazayeri, M. 2007. Some Trends in Web Application Development. In *2007 Future of Software Engineering* (May 23 - 25, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 199-213. DOI=<http://dx.doi.org/10.1109/FOSE.2007.26>

- Junqueira, D. C. and Fortes, R. P. 2004. VersionWeb: A Tool for Open Source Software Development Support. In *Proceedings of the Webmedia & La-Web 2004 Joint Conference 10th Brazilian Symposium on Multimedia and the Web 2nd Latin American Web Congress - Volume 00* (12 – 15 Outubro, 2004). LA-WEBMEDIA. IEEE Computer Society, Washington, DC, 65-67.
- Lerner, R. 1997. At the Forge: CGI Programming. *Linux J.* 1997, 33es (Jan. 1997), 10.
- Lerner, R. M. 2006. Introduction to Ruby. *Linux J.* 2006, 147 (Jul. 2006), 2.
- Reis, C. R. and Fortes, R. P. M. 2002. An Overview of the Software Engineering Process and Tools in the Mozilla Project. In: *Proceedings of Workshop on Open Source Software Development*, New Castle UK. v. 1. p. 162-182.
- Sasine, J. M. and Toal, R. J. 1995. Implementing the model-view-controller paradigm in Ada 95. In *Proceedings of the Conference on Tri-Ada '95: Ada's Role in Global Markets: Solutions For A Changing Complex World* (Anaheim, California, United States, November 05 - 10, 1995). TRI-Ada '95. ACM Press, New York, NY, 202-211. DOI= <http://doi.acm.org/10.1145/376503.376571>
- Stamey, J. and Richardson, T. 2006. Middleware development with AJAX. *J. Comput. Small Coll.* 22, 2 (Dec. 2006), 281-287.
- Thomas, D., Hansson D. H. 2006. “Agile Web Development with Rails” Pragmatic Bookshelf. Second Edition. 750 p. 2006.
- Williams, J. “Rails solutions: Ruby on Rails made easy.” New York, USA. Springer-Verlag, 2007.