CENTRO UNIVERSITÁRIO DO TRIÂNGULO
PRÓ-REITORIA DE GRADUAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Melhoria de Performance em Aplicações *Ruby on Rails*

Eduardo de Oliveira Vasconcelos

Uberlândia, Junho/2011.

CENTRO UNIVERSITÁRIO DO TRIÂNGULO
PRÓ-REITORIA DE GRADUAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Melhoria de Performance em Aplicações Ruby on *Rails.*

Eduardo de Oliveira Vasconcelos

Monografia apresentada ao Curso de Ciência da Computação do Centro Universitário do Triângulo - Unitri, como requisito básico à obtenção do grau de Bacharel em Ciência da Computação, sob a orientação do Prof. Marcos Alberto Lopes da Silva.

Uberlândia, Junho/2011.

Melhoria de Performance em Aplicações Ruby on *Rails*.

Eduardo de Oliveira Vasconcelos

Monografia apresentada ao Curso de Ciência da Computação do Centro Universitário do Triângulo - Unitri, como requisito básico à obtenção do grau de Bacharel em Ciência da Computação.

Marcos Alberto Lopes da Silva, Dr.
(Orientador)

XXXXX, Msc.
(Avaliador)

XXXXX, Msc.
(Avaliador)

Clayder Cristiam Coelho, Msc.
(Coordenador do Curso).

Uberlândia, Junho/2011.

Agradecimentos à
minha adorável mãe,
a responsável por tudo em minha vida.

RESUMO

Este trabalho tem como um dos objetivos, situar o leitor sobre o desenvolvimento web utilizando como ferramenta de apoio, o *framework*. Apresentar a linguagem *Ruby,* bem como suas particularidades, características e também o *framework Rails,* que o complementa, insere uma nova didática no desenvolvimento de aplicações *web* e interage harmoniosamente com a linguagem. O estudo visa atingir o objetivo principal de mostrar a importância do requisito performance e o quanto ela pode melhorar a iteração entre cliente-servidor em aplicações *web.* Para constatar e testar o ganho de performance foi criada uma gema para aplicações *Ruby on Rails* que insere melhorias e através dos seus resultados ficou comprovado que o objetivo principal do estudo foi alcançado. O método de investigação utilizado foi a pesquisa bibliográfica, ilustrada com a criação da gema.

Sumário

1 – INTRODUÇÃO

O aumento do número de computadores conectados a internet é um fato que se confirma por diversas pesquisas realizadas, como a realizada pelo Centro de Estudos sobre as Tecnologias da Informação e da Comunicação, disponível no site do Portal de Governo Eletrônico do Brasil, que comprova o aumento das compras e acesso a internet.

O número de usuários que utilizam a internet para realizar operações de compra, venda, consulta bancária e muitas outras obrigaram os administradores a construírem grandes sistemas performáticos em um tempo hábil cada vez menor. Estes sistemas se tornam importantes já que podem aumentar o lucro de uma determinada empresa, seu alcance comercial e transpor seus limites físicos. Para auxiliarem os administradores na construção destes sistemas, os desenvolvedores utilizam *softwares* de apoio chamados *frameworks* que otimizam o tempo de produção. Dentre estes *frameworks* é possível encontrar o *Rails* que implementa a topologia MVC (Modelo Visualização Controlador) na linguagem de programação *Ruby.*

O *Rails* é um *framework* que possui rotinas que auxiliam os desenvolvedores e administradores de uma forma ágil e fácil. Suas convenções facilitam a criação de sistemas completos para internet e podem diminuir o tempo total de desenvolvimento da aplicação. Porém assim como muitos *frameworks,* o *Rails* apesar de abrangente, não têm como um dos seus objetivos por padrão a melhoria de performance em suas aplicações. Apesar de possuir ferramentas para executar tais melhorias, o *framework Rails* não as insere por padrão deixando a cargo

do administrador do projeto estas inserções. A criação de grandes aplicações pode gerar, com o aumento do número de usuários, problemas de performance que comprometem o bom funcionamento do sistema ou até mesmo a sua indisponibilização. Diante disto, tornou-se objetivo principal deste projeto acadêmico, a criação de uma gema *Ruby,* para auxiliar os administradores em projetos de aplicativos Web, que desconhecem ou não realizam melhorias de performance em suas aplicações *Ruby* usando o *framework Rails*.

Uma gema *Ruby* funciona identicamente a um *plugin Ruby,* porém existe uma característica que difere os dois. A diferença é quanto a instalação, pois uma gema não necessita de nenhum software adicional para instalá-la. Já o *plugin* é instalado com a ajuda do controlador de versões *Git*. Para instalar a gema digita-se o comando `gem performance`.

Este projeto está sintetizado em cinco capítulos. O primeiro capítulo compreende esta introdução ao tema da monografia. O segundo capítulo apresenta uma visão geral dos assuntos relacionados com o objetivo principal deste projeto, contextualizando o leitor no desenvolvimento Web com a tendência atual de se contar com *frameworks*. O terceiro capítulo ilustra as principais características técnicas, pacotes e particularidades contidas na linguagem *Ruby* e no *framework Rails.* O quarto capítulo apresenta e discute as melhorias que serão abordadas e resolvidas na gema proposta bem como os resultados obtidos. A conclusão do projeto será apresentada no capítulo cinco. Além dos capítulos citados as fontes de pesquisa encontram-se nas referências bibliográficas.

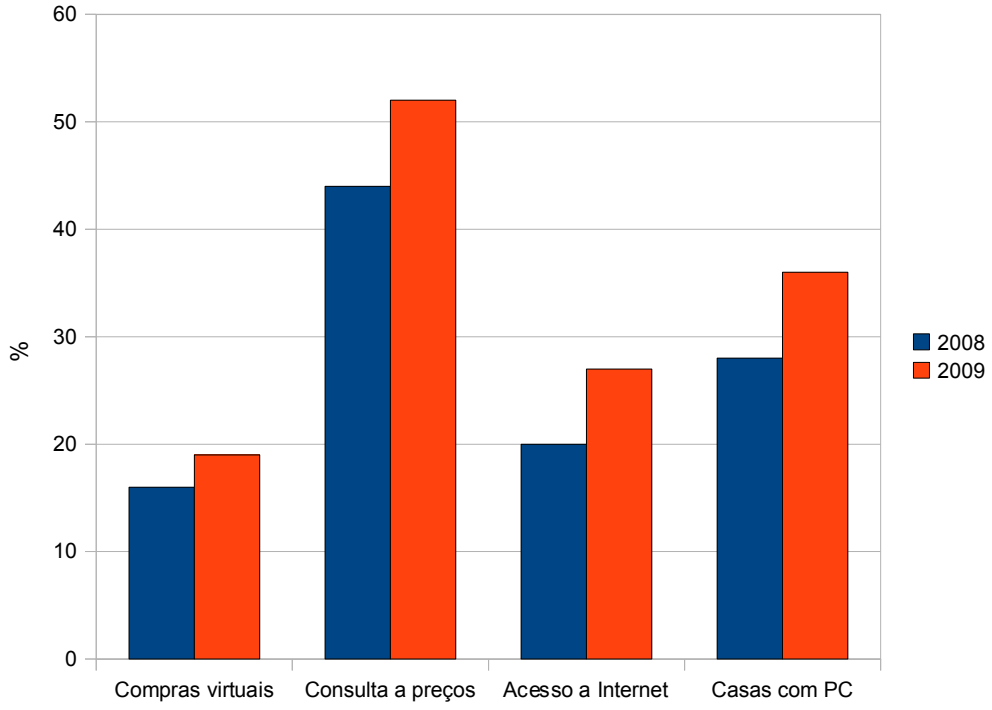
2. CRESCIMENTO DA WEB

Desde o começo da Internet, em meados dos anos 60, seus idealizadores não poderiam imaginar que uma ferramenta que era utilizada única e exclusivamente para interligar universidades, instituições de pesquisa e também órgãos militares pudesse se expandir tanto e que seus objetivos fossem rapidamente modificados para uma invenção de extrema importância no século XX. Sabe-se que essa expansão é constante e que por isso algumas mudanças em sua estrutura já estão sendo testadas a fim de atender a crescente demanda pelo seu acesso. Além de outras necessidades como segurança e escalabilidade, a necessidade de mudanças pode ser comprovada pela criação do IPv6 (Internet *Protocol version 6*, Protocolo da Internet versão 6) que é uma nova forma de endereçamento para os computadores conectados na mesma. A criação do protocolo visa aumentar a capacidade de computadores conectados na rede, dos atuais providos pelo protocolo IPv4 (internet *Protocol version 4*, Protocolo da Internet versão 4) para um número bem maior devido a sua capacidade diferenciada de endereçamento. (SMETANA, 2001).

As empresas que exploram este ambiente que aproxima virtualmente a todos notaram que o universo virtual é promissor e que o mesmo poderia aumentar seus lucros. Existem empresas que atuam apenas no mercado virtual desfazendo-se assim do modelo tradicional de negócio que envolve duas pessoas físicas, numa ponta o vendedor e na outra o comprador, além das empresas que utilizam os dois mercados, o virtual e o físico.

A razão para que estas empresas atuem cada vez mais no meio virtual é facilmente explicada, pois de acordo com a pesquisa realizada em 2009, nos domicílios brasileiros pelo Centro de Pesquisas sobre as Tecnologias da Informação e Comunicação houve um aumento significativo, como ilustrado no Gráfico 2.1, no número de computadores nos domicílios brasileiros. Além disso, constata-se também o crescimento no número de pessoas que possuem Internet em casa, consequentemente um aumento no número de pessoas que acessam a Internet e utilizam a mesma para consultarem preços e fazerem compras por motivos de comodidade, segurança e rapidez.

Gráfico 2.1: Resultados da pesquisa realizada em 2009.



Empresas investem grandes quantias em busca de sistemas que possam atender a todas as suas necessidades e suas expectativas. Tais sistemas demandam tempo para serem planejados, produzidos e as particularidades de cada modelo de negócio devem ser implementadas a fim de atender as regras de negócio cada empresa. Nota-se então que

desenvolvimento de sistemas para estas empresas tende a crescer na mesma proporção em que às mesmas são criadas.

2.1 - FRAMEWORKS

A fim de atender a crescente demanda de softwares que são projetados especialmente para a web são utilizadas vários tipos de linguagens de programação como o Java que é a linguagem mais utilizada na internet devido a uma série de qualidades como portabilidade, interoperabilidade, polimorfismo entre outras (NETO, 2007). Durante o processo de desenvolvimento de sistemas para internet, o programador se depara com rotinas que se repetem constantemente além de terem que cumprir curtos prazos para entrega de grandes aplicações. Sabendo disto, alguns desenvolvedores utilizam os *framework*s que são ferramentas responsáveis por automatizar o desenvolvimento e auxiliá-lo com tarefas corriqueiras e cansativas que demandam tempo e consequentemente podem atrasar a entrega do sistema. Os *framework*s possuem vários códigos de apoio, bibliotecas de código e através da reusabilidade de código reduzem os gastos na produção (IBM, 1997). Dentre outros objetivos também melhoram o desempenho das aplicações facilitando muito o trabalho do programador. A sua utilização não isenta o programador da necessidade de um conhecimento sobre todos os assuntos relacionados à produção de sistemas para Internet, como por exemplo, a interação de seu sistema com um sistema de gerenciamento de banco de dados onde todas as operações CRUD (*Create Read Update Delete* - Criar Ler Atualizar Deletar) devem estar implementadas. Os *framework*s são classificados quanto à forma de utilização e tipo estrutural. Suas formas de utilização se dividem em três*(*IBM,1997*)*:

1 - *Inheritance-focused* onde as funcionalidades e métodos podem sem alteradas ou estendidas através de *override* (sobreposição) de métodos. Estes *frameworks* são também conhecidos como *white frameworks*.

2 - *Composition-focused* que é o tipo onde nada poderá ser modificado pelo programador, pois os métodos já estão pré-fabricados e as funcionalidades já estão prontas no corpo do *framework* e o programador não consegue modificá-las. Conhecidos como *black frameworks*.
3 – Híbridos que possuem as características dos dois anteriores e são mais flexíveis e utilizados para a produção de sistemas. São chamados de *gray frameworks* e foram criados para obterem os benefícios dos dois tipos anteriores.

Suas formas estruturais assim como suas formas de utilização se dividem basicamente em três onde a escolha especificamente de cada uma depende do local onde o *framework* será utilizado. Em grandes sistemas pode-se encontrar várias formas. Suas formas são (IBM,1997):

1 – *Frameworks* de Suporte que disponibilizam todos os serviços idênticos ao da camada do sistema operacional como, acesso a arquivos, computação distribuídas, *devices drives* (Drives de dispositivos), etc.
2 – *Frameworks* de Integração são aqueles utilizados em aplicações distribuídas e componentes, visam aumentar o suporte a softwares de infra-estrutura que trabalham em ambientes distribuídos.
3 – *Frameworks* de aplicação que são mais usados no meio comercial. São essenciais para utilização em atividades específicas de um conjunto determinado das regras de negócio em que são aplicados.

2.3 - VANTAGENS E DESVANTAGENS DO USO DE UM FRAMEWORK

Um *framework* assim como qualquer software produzido possui vantagens e desvantagens, porém o que diferencia um bom software dos demais é a capacidade de seus pontos positivos sobressaírem sobre seus pontos negativos ou entre outras palavras que seus pontos negativos não

interfiram na operabilidade do mesmo. Basicamente um bom *framework* deve auxiliar a produção do sistema utilizando maximização de re-uso, fatorando qualquer aspecto comum entre aplicações, seja ele em código, análise, testes ou no *design*. Desta forma o programador poderá concentrar-se em tarefas específicas e na solução dos problemas de domínios.

Um bom *framework* deve exigir o mínimo de manutenção e possuir um código com menos defeito possível. Porém seu maior benefício é a redução dos custos da produção de um sistema, já que ele reduz o tempo de produção do mesmo (IBM,1997).

A construção de um bom *framework* é muito complexa, seu planejamento demanda tempo e isto é uma desvantagem. Outra desvantagem é com relação a manutenção, pois ela é muito custosa e pode comprometer a compatibilidade e consistência entre aplicações (IBM,1997).

2.5 – PERFORMANCE ENTRE APLICAÇÕES PARA WEB

O entendimento de performance está diretamente relacionado com a velocidade com que um programa pode responder a solicitações (SOUDERS, 2007). Mas em aplicações direcionadas para a internet, deve-se observar outros fatores que influenciam mais do que a diferença de performance entre as linguagens de programação. Isto não torna inviável a utilização de nenhuma linguagem, mas dependendo do objetivo específico da aplicação, algumas são mais indicadas do que outras. Entre os fatores que devem ser observados como: latência da rede, diferentes meios físicos, *hardware* diversos, versões diferentes entre *browsers* diferentes, sistemas operacionais, entre outras.

Dentre tantas diferenças pode-se utilizar uma forma bem simples de melhorar a performance das aplicações usando técnicas a fim de atingir o objetivo principal de melhoria da comunicação entre as máquinas

cliente e o servidor (EDIGER, 2007). Algumas destas técnicas são comuns a qualquer linguagem de programação e podem ser utilizadas por qualquer uma delas, outras são específicas e se aplicam apenas dentro de certa linguagem (SOUDERS, 2007). As duas formas de melhoria, ou seja na linguagem e na comunicação, serão aplicadas dentro da gema que será proposta como objetivo final desta monografia, na tentativa de melhorar a velocidade das aplicações que utilizem a linguagem *Ruby* juntamente com o framework *Rails* em seu desenvolvimento e também produção.

2.6 – CONCLUSÃO

A utilização de um *framework* durante todo o processo de produção de um software é de grande valia para um programador, pois ele auxilia no desenvolvimento de todas as etapas e pode reduzir significativamente o custo final do software. Sua facilidade de utilização é um ponto importante a ser levado em consideração, pois a realização de treinamento pode onerar ainda mais o custo do projeto. A escolha de um *framework* bem como suas vantagens, desvantagens e particularidades devem ser observadas com o intuito de agregar mais dinamismo e qualidade ao projeto. No próximo capítulo apresentaremos à linguagem *Ruby* e o *Rails,* sua forma de tratamento dos dados, implementação das características comuns e suas particularidades.

3 – O *FRAMEWORK RAILS*

O *framework Rails* possui muitas características, particularidades, vantagens e desvantagens, mas deve-se analisar primeiramente a linguagem de programação que ele complementa, neste caso a linguagem *Ruby*. O *Ruby* é uma linguagem interpretada e totalmente orientada a objetos e se caracteriza por uma tipagem dinâmica e forte, além de ser totalmente portátil (FULTON, 2006). Dentre outras características ela possibilita ao programador a inserção de código em classes nativas da linguagem com o objetivo de melhorar, adaptar ou particularizá-las (FULTON, 2006).

3.1 - A LINGUAGEM RUBY

O *Ruby* assim como qualquer outra linguagem de programação permite ao programador realizar operações para manipulações em arquivos, *threads* e *sockets*. A linguagem dá suporte à conexão e interação com vários sistemas de gerenciamento de banco de dados, como MySQL, PostGREE, Oracle, SQLite, etc (FULTON, 2006). Atualmente a linguagem *Ruby* está na versão 1.9.2 e foi tal versão utilizada nos exemplos que se seguem. O sistema operacional utilizado foi o *Snow Leopard*, mas como a linguagem é portátil, os exemplos abaixo poderão ser executados em qualquer sistema operacional. Seu *download* pode ser realizado através do site da linguagem. Suas declarações de dados são simples e abaixo exemplos para ilustrar e demonstrar suas

facilidades. A idéia é simplesmente apresentar os principais recursos da linguagem para que o entendimento do *Rails* seja facilitado.

IMPORTAÇÃO DE ARQUIVO EXTERNO → Se programador necessitar de utilizar alguma funcionalidade presente em algum arquivo externo, basta ele utilizar o comando `require` dentro da classe *Ruby* que necessite da funcionalidade (COLLINGBOURNE, 2008), como visto na Figura 3.1.

```
1  class MyClass
2
3  require "arquivo"
4
5  attr_accessor :alt #Campo para escrita e leitura (get/set)
6  attr_reader :greeting #Apenas para leitura (get)
7  attr_writer :num_pallets  #Apenas para escrita (set)
8
9    def m1 #Este método é publico.
10   end
11
12 protected
13   def m2 #Este método é protegido.
14     puts "Tente Novamente"  if contador > 10
15     tentativa == 3 ? puts "Você Perdeu" : puts "Entre com um número"
16   end
17
18 private
19   def m3 #Este método é privado.
20         while alt < 100 and numero <= 30
21            pallet = next_pallet()
22            alt += pallet.weight
23            num_pallets +=1
24          end
25   end
26 end
```

Figura 3.1: Exemplo de classe, métodos, estruturas de controle *Ruby*.

MÉTODO → Um método é declarado através da palavra reservada `def` e logo após o nome do método e caso necessite de parâmetros, os mesmos devem ser declarados após o nome do método e entre parênteses. O que delimita o final de um método é a palavra reservada

end (COLLINGBOURNE, 2008). Como exemplificado na Figura 3.1. Por convenção o nome do método é iniciado com letras minúsculas.

CLASSE → Uma classe no *Ruby* necessita da palavra reservada `class` seguido de um nome (COLLINGBOURNE, 2008), que deve ser iniciado, por convenção, com uma letra maiúscula, como demonstrado na Figura 3.1.

PRIVADO E PROTEGIDO → No *Ruby* os métodos são públicos por padrão, logo qualquer usuário pode executá-los. O *Ruby* possui uma forma de proteger e fazer com que os métodos de uma classe sejam privados ou protegidos. A diferença é que métodos privados podem ser utilizados somente por objetos da mesma instância. Já o método protegido além de também serem utilizados pela mesma instância, podem ser chamados por subclasses (COLLINGBOURNE, 2008). Um exemplo é mostrado na Figura 3.1. Desta forma o programador pode restringir a utilização dos métodos e aumentar a segurança do seu software.

MÉTODOS *GET* E *SET* NO *RUBY* → O *Ruby* disponibiliza uma facilidade para o programador no que se diz respeito aos métodos *get* e *set.* Eles são bastante utilizados em linguagens como o Java. Porém o *Ruby* propõe palavras reservadas para solucionar este problema onde: `attr_reader` é utilizado para leitura, `attr_writer` para escrita e `attr_accessor` para leitura e escrita (COLLINGBOURNE, 2008), como visto no exemplo na Figura 3.1.

ESTRUTURAS DE CONTROLE → O *Ruby* possui estruturas para controlar o fluxo de execução do programa. Como nos métodos a palavra reservada *end* delimita o final destas estruturas, como está exemplificado na Figura 3.1.

ARRAY E HASHES → Tanto os *arrays* quanto os *hashes* no *Ruby* são coleções indexadas que são acessíveis através de uma chave. A diferença entre eles é que no *array* a chave é um número inteiro e nas *hashes* a chave pode ser qualquer objeto. Tal diferença entre eles gera nos *arrays* um acesso mais eficiente a seus elementos enquanto que nas *hashes* o programador ganha em flexibilidade (COLLINGBOURNE, 2008). Uma característica da linguagem *Ruby* é que tanto um *array* quanto um *hash* pode conter tipos variados de dados em sua composição. Desta forma qualquer um dos dois pode conter números inteiros, *strings,* ou números reais, por exemplo (COLLINGBOURNE, 2008). O *Ruby* possui atalhos para a criação de *arrays*, como visto na Figura 3.2.

```
1  a = [ 1, 'cat' , 3.14 ] # array com três elementos
2  a[0] # acessando o primeiro elemento (1)
3  a[2] = nil # alterando o terceiro elemento
4
5  a = [ 'ant' , 'bee' , 'cat' , 'dog' , 'elk' ] # é o mesmo que
6  a = %w{ ant bee cat dog elk }
```

Figura 3.2: Exemplo de um *Array.*

EXPRESSÕES REGULARES → A utilização de expressões regulares para validação de dados é largamente utilizada em linguagens de programação. Sua declaração é feita de duas formas: Colocando a string entre duas barras ou utilizando `%r` seguido à cadeia de caracteres que será validada (COLLINGBOURNE, 2008), como exemplificado na Figura 3.3.

```
1  if line =~ /P(erl|ython)/
2    puts "There seems to be another scripting language here"
3  end
```

Figura 3.3: Exemplo de Expressão Regular.

BLOCOS E *LAMBDA* → Os blocos ou *closures* podem ser declarados de duas formas. Se a código possuir apenas uma linha, o mesmo deverá ser colocado entre chaves, mas se possuir mais linhas será alocado entre os

delimitadores *do/end.* “A palavra ‘lambda’ fecha o código entre chaves como um objeto, num bloco, uma instância da classe Proc. Este objeto responde ao método ‘call’.” (AKITA, 2006). Como na Figura 3.4.

```
1  greet { puts "Hi" }
2  3.times { print "Ho! " } #=> Ho! Ho! Ho!
3  c = lambda { |i| puts i }
4  c.call(1) # => 1
5  c.call(2) # => 2
```

Figura 3.4: Exemplo de utilização de um *bloco, lambda* e um *interador.*

EXCEÇÕES → O *Ruby* trata suas exceções como objetos. Elas são declaradas entre as palavras reservadas *begin* e *end.* Utiliza-se cláusulas de resgate para identificar e tratar tais exceções como no exemplo ilustrado na Figura 3.5.

```
1  begin
2    content = load_blog_data(file_name)
3    rescue BlogDataNotFound
4      STDERR.puts "File #{file_name} not found"
5    rescue BlogDataFormatError
6      STDERR.puts "Invalid blog data in #{file_name}"
7    rescue Exception => exc
8      STDERR.puts "General error loading #{file_name}: #{exc.message}"
9  end
```

Figura 3.5: Exemplo de um tratamento de exceção.

DOCUMENTAÇÃO → O acesso a documentação do *Ruby* pode ser feita através de linha de comando, digitando a palavra reservada *ri* seguida da classe e o método que deseja informação. Caso a classe contenha mais de uma documentação para o mesmo método, o *Ruby* mostrará as opções. A linguagem *Ruby* possui também um *site* onde o programador pode acessar toda a documentação da linguagem.

3.2 – O FRAMEWORK RAILS

O *Rails* é um *framework* de código aberto e desde sua disponibilização em julho de 2004, através de uma versão incompleta, ele obteve rapidamente a ajuda de muitos desenvolvedores até o lançamento da versão final 1.0, em dezembro de 2005. Para instalar o *Rails* em qualquer sistema basta apenas digita o comando *gem install rails* no prompt. É necessário estar conectado a internet, e após este comando o *Rails* será instalado juntamente com outros pacotes, na forma de gema, que serão explicados individualmente a seguir, mas os principais são (AKITA, 2006):

ActionPack: Pacote responsável por controlar o fluxo de ações no aplicativo entre as partes da *View* e *Controller*. Ele gerencia o redirecionamento a páginas e chamadas aos métodos (AKITA, 2006).

ActiveRecord: Responsável pela interação e persistência entre a aplicação e o banco de dados. Este pacote faz o mapeamento das tabelas, bem como facilita as operações no banco (TATE, 2006).

ActiveSupport: Administra tudo aquilo que não é gerenciado nos pacotes anteriores. É responsável por gerenciar outras bibliotecas, dependências, recursos avançados de *breakpoint* em tempo de execução, cachê, *plugins*, etc (AKITA, 2006).

Rails: É o *framework* propriamente dito e faz a junção de todos os pacotes que são baixados com ele. Além disso, ele disponibiliza uma série de geradores, *templates* para geração de código e tarefas do pacote *rake (AKITA, 2006)*.

Após a instalação de todos os pacotes necessários para a utilização do *Rails*, basta uma linha de comando para a criação de uma

aplicação, onde *rails* e *new* são palavras reservadas e app é o nome do novo sistema que será criado. *O Rails* cria uma série de diretórios e arquivos pré-configurados que serão utilizados durante todo o processo de criação do aplicativo. Os diretórios são (AKITA, 2006):

- ***app*** → Detém todo o código referente à aplicação em particular. Aqui são ramificados os componentes do MVC, Modelo-Visualização-Controlador. Ele é subdividido em três subdiretórios. Que são eles:

- ***app/controllers*** → Possui todos os controladores responsáveis por processar as ações solicitadas através das *views* .

- ***app/models*** → Possui os modelos da aplicação. Neles colocam-se as particularidades de cada classe de dados.

- ***app/views*** → É o diretório onde se aloca as páginas que serão expostas no navegador. Toda ação solicitada nas *views* serão processadas no controlador.

- ***app/views/layouts*** → Possui os arquivos modelo que serão utilizados pelas *views*. Neste diretório pode-se alocar um padrão comum que será utilizado por toda a aplicação. As *views* identificam seu conteúdo e o renderizam como modelo para as que não possuam um *layout* padrão.

- ***app/helpers*** → São ajudantes que auxiliam os controladores. Caso exista a necessidade de alguma função seja utilizada nas *views,* aloca-se tais procedimentos nos ajudantes e assim pode-se utilizá-los.

- ***config*** → Possui os arquivos para a configuração do ambiente *Rails*, como conexão no banco de dados e mapeamento de rotas.

- ***db*** → Possui todos os arquivos relacionados ao banco de dados do sistema, como as migrações criadas, tabelas e também todas as modificações que foram feitas no banco .

- ***doc*** → Este diretório é onde ficará a documentação da aplicação caso ela seja gerada através do comando rake doc:app.

- ***lib*** → Bibliotecas específicas e externas que contem funcionalidades que não estão presentes nos controladores, modelos ou ajudantes.

- ***public*** → Este diretório é publico e está disponível para todo o servidor, bem como todos os arquivos padrões que serão carregados. Nele coloca-se as imagens, *javascripts* e folhas de estilos.

- ***script*** → *Scripts* de ajuda para automatização e criação dentro da aplicação.

- ***test*** → Os testes unitários e funcionais são alocados neste diretório.

- ***vendor*** → As bibliotecas não nativas do *Rails* são alocadas neste diretório. Possui um subdiretório que inclui os *plugins* instalados.

3.3 – RAILS UM FRAMEWORK MVC

O *Rails* é um *framework* orientado a objetos do tipo MVC (Modelo-Visualização-Controlador) pois ele segue a metodologia que garante a divisão da aplicação em três componentes, e com isso cada um possui apenas uma responsabilidade. O gerenciamento das interfaces é feito pelo *ActionView.* Já o controle de fluxo da aplicação é feito pelo *ActionController.* Os dois são pacotes integrantes do *ActionPack.* As

regras de negócio são tratadas e implementadas pelo *ActiveRecord.* Todos estes componentes serão explicados mais detalhadamente neste capítulo. O padrão MVC (Modelo-Visualização-Controlador) torna a manutenção mais fácil, pois cada componente pode ser mantido separadamente não interferindo no funcionamento dos outros componentes (NETO, 2007).

3.4 – ACTIVERECORD

O *ActiveRecord* é o pacote que é instalado como uma gema juntamente com o *Rails* e é responsável por fazer o mapeamento objeto-relacional e implementar as regras de negócio da aplicação. O *ActiveRecord* é responsável por executar tarefas como conexão ao banco de dados, o mapeamento das tabelas e a manipulação de seus dados. O mapeamento utilizado pelo *ActiveRecord* é realizado de uma forma padrão onde tabelas são mapeadas através de classes, seus registros são mapeados através de objetos e colunas são acessadas através de atributos destes objetos (FERNANDEZ, 2010). A Figura 3.6 mostra um exemplo de manipulação de registro, onde localiza-se um determinado registro, executa a modificação e finalmente salva-se o registro.

```
1  class Pedido < ActiveRecord::Base
2  end
3
4  pedido = Pedido.find(5)   #localiza e seleciona o pedido id=5.
5  pedido.name = "Eduardo" #modifica o nome do comprador do pedido.
6  pedido.save       #salva o registro
7
```

Figura 3.6: Exemplo de uma classe *ActiveRecord* e algumas operações.

Não há necessidade de código dentro da classe Pedido para realizar operações, já que a classe Pedido herda da sua classe-pai neste caso o *ActiveRecord* todas as suas características, procedimentos e

funções (TATE, 2006). Todas estas facilidades tornam a manipulação do banco de dados mais flexível e agradável.

3.4.1 – RELACIONAMENTO ENTRE TABELAS

Grande parte das aplicações feitas para a internet possui um número considerável de tabelas, pois elas são responsáveis por armazenar vários tipos de dados, que vão desde um simples cadastro de clientes a pedidos complexos com dados específicos como horários de entrega e número de cartões de créditos. Os relacionamentos entre tabelas são comuns, criam elos entre elas e colocam na prática o modelo entidade relacional do banco de dados. A forma de implementar os relacionamentos utilizando o *ActiveRecord* é simples, intuitiva e fácil. As opções são as seguintes (TATE, 2006):

Relacionamento de 1 para 1: A classe detentora do objeto unitário deve conter as palavras reservadas *has_one* (possui um). Enquanto que a classe pertencente deve conter em sua classe a palavra reservada *belongs_to* (pertence a).
Relacionamento de 1 para muitos: A classe detentora da outra classe deve conter as palavras reservadas *has_many* (possui muitos). Enquanto que a classe pertencente deve conter a em sua classe a palavras reservadas *belongs_to* (pertence a).
Relacionamento de muitos para muitos: Neste caso, ambas as classes pertencem e detém uma a outra. Desta forma, ambas recebem as mesmas palavras reservadas *has_and_belongs_many* (possui e pertence a muitos).

Todas as declarações devem conter após as palavras reservadas o caractere dois pontos seguidos do nome da classe. Caso a classe seja detentora de outra que contenha muitos objetos, seu nome será colocado no plural como exemplificado na Figura 3.7.

```ruby
class Pedido < ActiveRecord::Base
   has_many :items #Classe Item
   belgons_to :cliente
   has_and_belongs_many :categories #Classe category
   has_one :invoice
end
```

Figura 3.7: Exemplo de relacionamentos numa classe.

3.4.2 – VALIDAÇÕES

As validações restringem erros durante operações, aumentam a segurança ou limitam um determinado intervalo de um campo específico (FERNANDEZ, 2010). Existe uma série de opções de validações que podem ser incorporadas dentro do modelo. Presente na Figura 3.8, alguns exemplos dos tipos de validações.

- validates_acceptance_of → Verifica se uma *checkbox* está selecionada. Um exemplo de utilização pode ser feito quando há necessidade de aceitação de algum termo.
- validates_presence_of → Obriga o usuário a preencher o campo especificado não autorizando que o mesmo fique em branco ou nulo.
- validates_numericality_of → Verifica se o digito que foi digitado no campo é um numero, seja ele real ou inteiro. O programador pode escolher entre as opções para que restrinja inclusive faixa que será alocada pelo usuário e o tipo de numero que será permitido.

```ruby
class Pedido < ActiveRecord::Base
  validates_acceptance_of :termo, :message => "Termo não assinalado."
  validates_presence_of :nome, :message => "Nome não pode ser branco."
  validates_numericality_of :quantidade, :only_integer => true,
  :greater_than => 0, :message => "Quantidade deve ser maior que 0."
end
```

Figura 3.8: Exemplo de validações.

3.4.3 – CALLBACKS

O *ActiveRecord* também possui algumas *callbacks* (chamadas do sistema) e tem como objetivo de controlar o ciclo de vida dos objetos do modelo quando os mesmos são criados, atualizados ou deletados. O *ActiveRecord* disponibiliza 20 *callbacks,* sendo que 18 são usadas antes e depois de alguma operação no banco de dados e 2 que são usadas em outros casos específicos (FERNANDEZ, 2010).

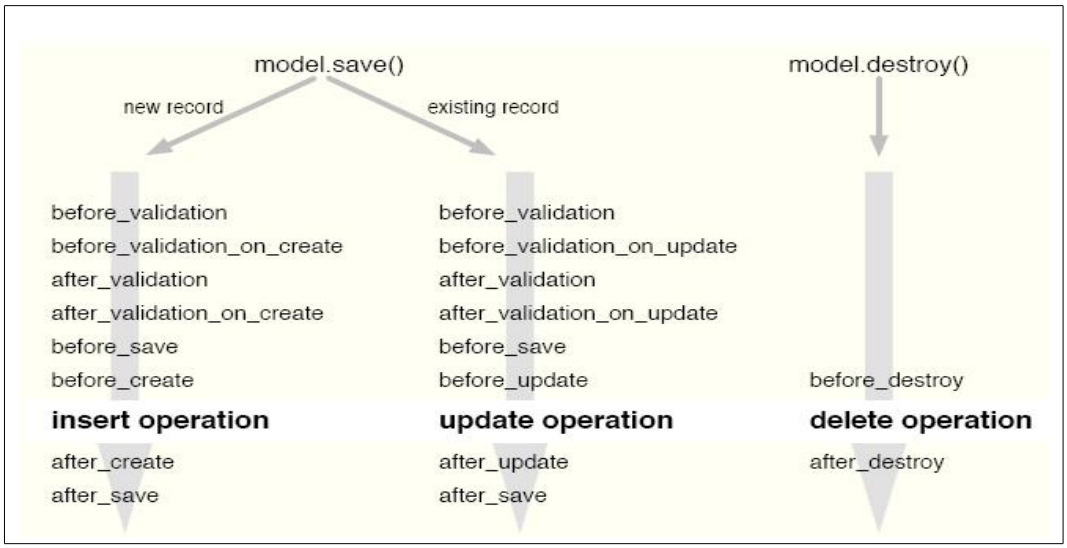


Figura 3.9: Clico de vida das *callbacks* do *ActiveRecord*.

A utilização das *callbacks* complementa as validações e observam o comportamento de objetos do modelo. Na Figura 3.9, estão ilustradas todas as 18 *callbacks* disponíveis no *ActiveRecord*.

```
1  class Pedido < ActiveRecord::Base
2       after_update :save_itempedidos #Callback
3
4       def save_itempedidos
5            itempedidos.each do |item|
6              if item.should_destroy?
7                   item.destroy
8              else
9                   item.save(false)
10             end
11           end
12      end
13 end
```

Figura 3.10: Exemplo de utilização de *callbacks*.

Para executar alguma ação utilizando uma *callback,* cria-se o procedimento com o código a ser executado, após isso, indica-se o nome da *callback* seguido de dois pontos e o nome do procedimento que contem o código que será executado (FERNANDEZ, 2010), como no exemplo na Figura 3.10.

3.5 – ACTION PACK

O *Action Pack* é o pacote que contêm os dois módulos restantes da metodologia MVC, *model* (modelo), *view* (visualização) e *controller* (controlador) usada pelo *Rails* para a criação de aplicações web*. O model* é tratado pelo módulo *ActiveRecord*, a *view* e o *controller* possuem seus respectivos módulos, *ActionView* e *ActionController*. Os dois juntos são responsáveis por receberem as requisições vindas do *browser,* processá-las internamente e entregar as repostas ao usuário (AKITA, 2006). Nas próximas subseções os dois pacotes serão tratados separadamente, apesar de trabalharem de uma forma unificada dentro do *framework Rails.*

3.5.1 – ACTIONCONTROLLER

As requisições oriundas do *browser*, são processadas e suas respostas enviadas ao *browser*. A aplicação utiliza o mapeamento da requisição para extrair seus parâmetros e processá-los. O arquivo responsável por conter todas as informações sobre as rotas é o `routes.rb` dentro do diretório `config` da aplicação (AKITA, 2006). Ele possui as seguintes linhas padrão, que são mostradas na Figura 3.11, para a definição da forma como as requisições serão processadas.

```
1 ActionController::Routing::Routes.draw do |map|
2   map.connect ':controller/:action/:id'
3   map.connect ':controller/:action/:id.:format'
4 end
```

Figura 3.11: Mapeamento de rotas no *routes.rb*

A regra acima é mapeada e toda requisição feita é processada e separada em três parâmetros: o primeiro parâmetro após o nome do site é o controlador, seguido da ação que ele executará e finalmente o número de identificação do objeto que está recebendo a ação. A configuração do arquivo `routes.rb` pode ser acrescida de características específicas, como na Figura 3.12, com a utilização de expressões regulares para validações.

```
1  map.connect "blog/:year/:month/:day" ,
2    :controller => "blog" ,
3    :action => "show_date" ,
4    :requirements => { :year => /(19|20)\d\d/,
5    :month => /[01]?\d/,
6    :day => /[0-3]?\d/},
7    :month => nil
```

Figura 3.12: Rotas com validação usando expressão regular.

O *Rails* suporta o padrão REST, *Representational State Transfer*, em suas aplicações (FERNANDEZ, 2010). Para isto basta o programador utilizar em seu mapeamento de rotas a palavra *resources* e o *Rails* criará uma série de rotas RESTful (FERNANDEZ, 2010), como na Figura 3.13.

```
1 ActionController::Routing::Routes.draw do |map|
2   map.resources :articles
3 end
```

Figura 3.13: Mapeamento REST.

Cada classe dentro da aplicação é mapeada dentro de seu respectivo controlador. Este arquivo é responsável por identificar e processar todos os parâmetros das requisições externas vindas das *views* dentro da aplicação. Dentro de cada *controller (*controlador), as *actions* (ações) executam determinada operação, como visto na Figura 3.14.

```ruby
class PedidosController < ApplicationController
  def show
    @pedido = Pedido.find(params[:id])
  end

  def index
    @pedidos = Pedido.paginate :page => params[:page], :per_page => 5
  end

  def new
    @pedido = Pedido.new(:cliente => Cliente.find(params[:cliente_id]))
    1.times {@pedido.itempedidos.build}
  end
end
```

Figura 3.14: Exemplo de um controlador e suas respectivas *actions.*

3.5.2 – ACTIONVIEW

Após o mapeamento de uma requisição, o controlador a processa dentro de sua respectiva *action.* Finalizado o processamento por parte do controlador, ele envia à resposta da requisição feita, apresentado-a no *browser*. O *ActionView* é o módulo responsável por controlar todas as funcionalidades para a apresentação da resposta e o envio dos parâmetros da requisição. Ele compõe a metodologia MVC, pois controla as *views* (visualizações).

```erb
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <%= csrf_meta_tag %>
    <%= render 'layouts/stylesheets' %>
  </head>
  <body>
    <div class="container">
        <%= render 'layouts/header' %>
            <section class="round">
                <%= yield %>
            </section>
    </div>
  </body>
</html>
```

Figura 3.15: Exemplo de uma *view.*

Estas respostas são renderizadas sob a forma de *templates* que comumente são usados na *web,* eles possuem os formatos *HTML, XML* ou *JavaScript*. As *views* possuem uma mistura de texto e código *Ruby* introduzindo mais dinamismo dentro da *template*, como mostrado na Figura 3.15. Existe uma importante ligação entre o controlador e a sua respectiva *template,* pois suas variáveis, alguns objetos especiais e métodos do controlador estão acessíveis e podem ser usados pela sua *template (FERNANDEZ, 2010).*

3.6 – MIGRAÇÕES

Durante a criação de um sistema, por várias vezes existe a necessidade de modificações em seu código. Os motivos para tais modificações vão desde a inserção de novas funcionalidades no sistema, mudança nas regras de negócio ou até simples erros de codificação. Isto envolve operações no banco de dados, como inserção de novas tabelas, novas colunas, relacionamentos e outras operações que podem afetar todo o sistema. Para facilitar tais procedimentos o *Rails* utiliza as *migrations (migrações)* que automatizam todo o processo.

```
1  class OpColunaEmail < ActiveRecord::Migration
2    def self.up
3      rename_column :pedidos, :e_mail, :customer_email
4      change_column :pedidos, :tipo_pedido, :string
5      Produto.update_all("preco = preco * 100" )
6      change_column :produtos, :preco, :integer
7    end
8    def self.down
9      change_column :pedidos, :order_type, :integer
10     rename_column :pedidos, :customer_email, :e_mail
11     Produto.update_all("preco = preco / 100.0" )
12   end
13 end
```

Figura 3.16: Renomeação e modificação do tipo de uma coluna utilizando *migration.*

Na criação de uma migração, o comando criará um arquivo que conterá os dados da migração recém-criada (FERNANDEZ, 2010). Nele podem-se alocar vários tipos de modificação no banco de dados como exemplificado na Figura 3.16.

As migrações também podem ser utilizadas para modificar definição de índices e chave primária, como exemplificado na Figura 3.17, já que o *Rails* cria por padrão uma coluna do tipo inteira auto incrementável chamada `id` em cada tabela criada no banco de dados.

```
1 class NomeDaMigracao < ActiveRecord::Migration
2   def self.up
3     add_index :nome_da_tabela, :nome_da_coluna
4   end
5   def self.down
6     remove_index :nome_da_tabela, :nome_da_coluna
7   end
8 end
```

Figura 3.17: Modificação o índice de uma determinada tabela.

Os tipos de dados suportados pelas migrações possuem diferenças e elas devem ser observadas de acordo com o sistema de gerenciamento de banco de dados que será utilizado pelo software. O aspecto mais importante da utilização das *migrations* é que não há necessidade de fazer código em SQL para realizar modificações ou operações no banco de dados pois tudo é feito pelo *ActiveRecord (TATE, 2006)*.

3.7 – ACTIVE SUPPORT

O *Active Support* é o pacote responsável por dar suporte a todos os componentes do *Rails*, ele possui uma série de bibliotecas que estão disponíveis para estes componentes. Além disto, ele tem o propósito de estender as bibliotecas nativas do *Ruby* inserindo procedimentos que aumentam internamente o poder e aumentam as funcionalidades da linguagem. Ele implementa no *Rails* o que a linguagem *Ruby* não tratou

ou não implementou. São funcionalidades que auxiliam o programador com cálculos e conversões que antes necessitavam de criação de métodos específicos ou utilização de classes para tais (EDIGER, 2007).

3.8 – CONCLUSÃO

A linguagem *Ruby* possui novos conceitos e particularidades que adicionam uma nova forma conceitual ao universo das linguagens de programação. O capítulo ilustrou como o *Ruby* trata seus dados onde todos são objetos. Porém suas peculiaridades se tornam interessantes devido à intenção do seu criador de tornar todo o processo de programação e produção de softwares o mais intuitivo e simples possível.

Neste capítulo algumas das particularidades e funcionalidades do *framework Rails* foram apresentadas bem como suas técnicas de implementação das regras de negócio utilizando a metodologia MVC, além da forma como ele estende algumas funcionalidades da linguagem *Ruby* e como gerencia todos os seus pacotes.

A utilização do *framework Rails* para o desenvolvimento de um sistema completo direcionado para web é possível já que ele atende a todas as exigências necessárias para a criação do mesmo. No próximo capítulo será apresentada uma gema com o objetivo de tentar melhorar a performance em aplicações que utilizem o *framework Ruby on Rails,* como ferramenta de desenvolvimento.

4 - MELHORIA DA PERFORMANCE EM APLICAÇÕES *RUBY ON RAILS*

A utilização de uma linguagem mais performática influencia diretamente na capacidade de resposta de algumas aplicações, porém o que realmente interessa em aplicações para a *web* são: uma codificação eficiente, melhorias na iteração cliente-servidor, a melhoria de rotinas que auxiliem os *browsers* no carregamento das páginas de uma forma mais rápida e no lado do servidor, na melhoria no tempo de resposta às requisições efetuadas pelos *browsers* do cliente. Há divergências entres desenvolvedores que acreditam que melhorias devem ser feitas somente no lado do servidor ou vice-versa. Existem dezenas ou até milhares de autores que tratam sobre o assunto, tanto de forma genérica quanto específica para o *Ruby on Rails*. Diante de tantos procedimentos a criação de uma gema para auxiliar desenvolvedores a melhorar a performance no servidor e no cliente tornou-se foco desta monografia.

4.1 - MELHORIAS DE PERFORMANCE NO CLIENTE

Alguns autores insistem que apenas melhorias no lado do cliente resolvem o problema de performance, pois uma rápida resposta a uma requisição feita pelo mesmo mostrará o quanto seu sistema é eficiente a requisições. Entende-se por requisição desde o carregamento total dos dados de uma página, como imagens, processamento de *javascripts*, regras de estilo e outros elementos (SOUDERS, 2009).

Com base nestas informações percebe-se que a gema que será produzida terá como um dos seus objetivos, a procura por melhorias que ainda não foram feitas pelo desenvolvedor no lado do cliente e consiga efetuá-las de forma automática inserindo todas as suas dependências, utilizando *softwares* de apoio e procedimentos. As melhorias que serão feitas na aplicação são as seguintes:

1 – Junção de arquivos

Alguns autores definem regras para aumentar a performance no lado cliente e de acordo com *Steve Souders,* elas podem ser divididas em 14 regras básicas. Agrupando-as pode-se resolvê-las de uma forma mais prática. A primeira regra e mais importante é a minimização de requisições (*SOUDERS, 2007)*, pois quanto maior o número de requisições mais tempo será desprendido para fazer o *download* destes arquivos e o custo para processá-los será maior. Pode-se resolver este problema unificando arquivos como folhas de estilos (CSS), *JavaScripts* e imagens. Porém internamente no escopo do *Rails* a compressão de imagens não está implementada.

Desta forma, cabe ao programador através de configurações em seu CSS utilizar esta técnica. A deleção de arquivos duplicados melhora a performance porque evitará requisições duplicadas e desnecessárias. Outra regra é a redução dos arquivos CSS e *JavaScript* com a retirada de comentários, quebra de linha e espaços em branco. Retirando tais caracteres que não são necessários, haverá uma diminuição no tamanho do arquivo a ser baixado. Para resolver as regras citadas obteve-se auxílio da gema chamada *Jammit,* que fará a compressão e redução dos arquivos CSS e *JavaScript.* Ela e suas dependências serão instaladas automaticamente e configuradas através da gema proposta. Anteriormente a execução da gema *jammit* cada arquivo CSS ou *javascript* será baixado separadamente como visto na Figura 4.1.

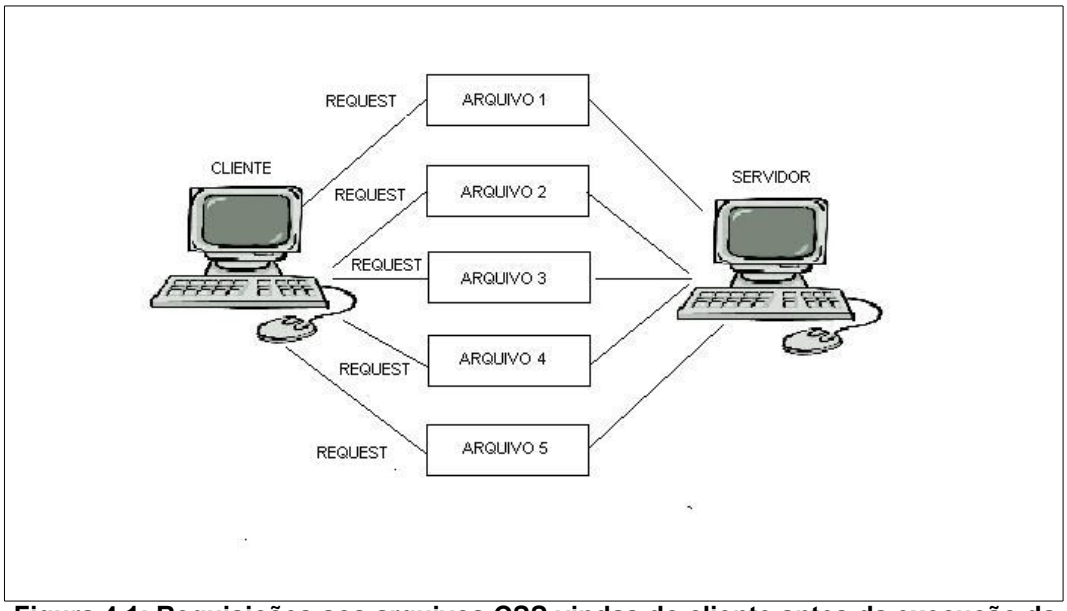


Figura 4.1: Requisições aos arquivos CSS vindas do cliente antes da execução da gema *jammit.*

Para realizar as operações o procedimento chamado `join_cssjs` fará a compactação e redução dos arquivos CSS e *JavaScript* presentes nos diretórios `public/javascripts` e `public/stylesheets` da aplicação.

```
27  make_datas
28  system("jammit")
29  p "Making procedures into app helper"
30  dir_helper =  "#{Rails.root}/app/helpers/"
31  File.open("#{dir_helper}application_helper.rb","a") do |read_helper|
32    read_reader.readlines().each do |line|
33      if line =~ /ApplicationHelper/ then
34        read_helper.puts "module ApplicationHelper\n\n\tdef stylesheets(*files)
35        \n\tcontent_for(:stylesheets)  { stylesheet_link_tag(*files) }\nend\n\n
36        def javascripts(*files)\n\tcontent_for(:javascripts)
37        { javascripts_link_tag(*files)}\nend"
38      else
39        read_helper.puts "#{line}"
40      end
41    end
42  end
```

Figura 4.2: Procedimento *join_cssjs.*

Ele fará com que os tipos de arquivos citados anteriormente fiquem reduzidos a um por tipo, sendo um arquivo CSS e outro *javascript*. O corpo principal do procedimento está presente no código da Figura 4.2.

Criará-se o arquivo de configuração *assets.yml* que armazenará o *namespace,* será referenciado dentro das *views* da aplicação e conterá o caminho dos arquivos que serão comprimidos e reduzidos, como mostrado na Figura 4.3. Em seguida insere-se automaticamente dentro da *view* principal da aplicação, uma linha de referência ao *namespace javascript* e outra com referência ao CSS.

```
71 File.new("assets.yml", "w") do |file|
72   file.puts("embed_assets: on\njavascripts:\n\t
73   workspace:\n\t\t - public/javascripts/*.js")
74   file.puts("\nstylesheets:\n\t
75   workspace:\n\t\t - public/stylesheets/*.css")
76 end
```

Figura 4.3: Procedimento que cria o arquivo *assets.yml.*

Após a execução da gema *jammit* haverá uma redução dos arquivos CSS e *javascript* bem como uma junção dos mesmos fazendo existir apenas uma requisição a este arquivo, como visto na Figura 4.4.

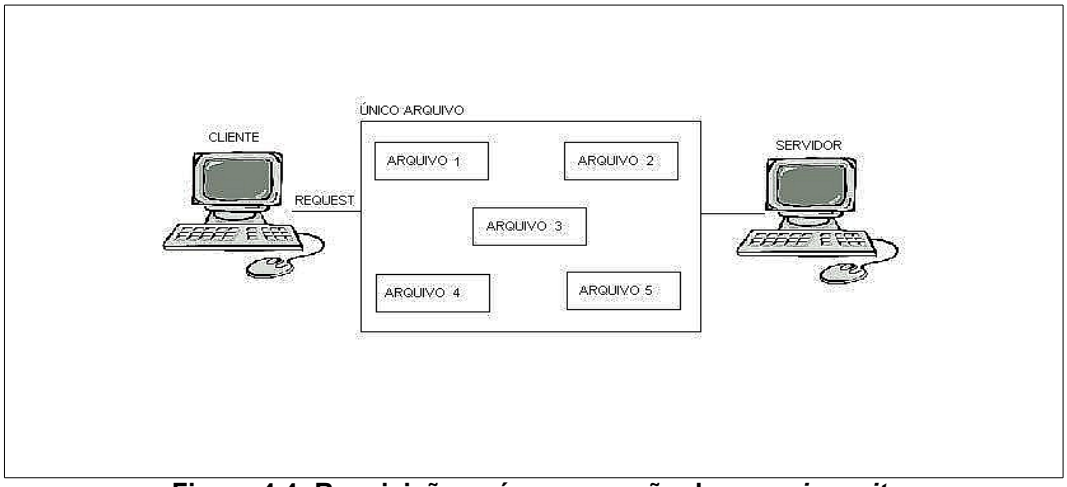


Figura 4.4: Requisição após a execução da gema *jammit.*

2 – Compressão de arquivos e *cachê* cliente

A compressão de arquivos e *cachê* utilizará recursos do servidor Apache. Esta melhoria está relacionada ao servidor *web*, podendo ser feita em qualquer aplicação independente de linguagem de programação.

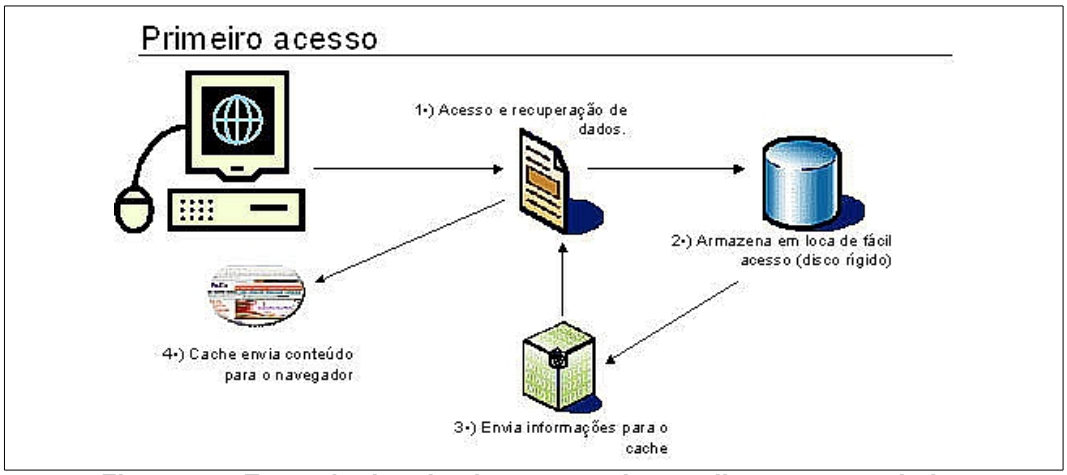


Figura 4.5: Exemplo de primeiro acesso de um cliente a um *website*.

Após o primeiro acesso, os arquivos ficarão em cache com duração de um mês e serão baixados quando sofrerem alterações ou o prazo expire como na Figura 4.6.

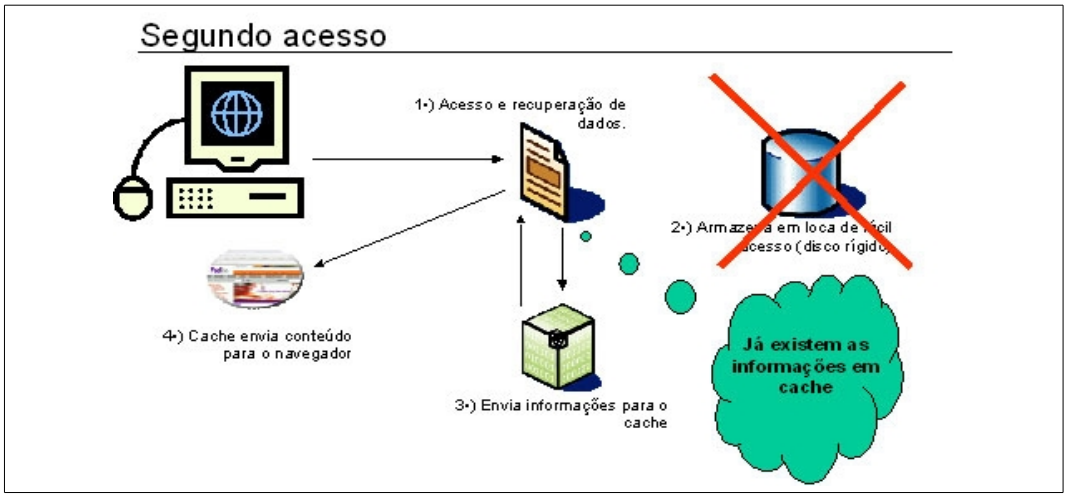


Figura 4.6: Segundo acesso numa máquina cliente que foi cacheada.

O procedimento `configs_apache` será o responsável por inserir um arquivo *.htaccess* dentro do diretório raiz da aplicação com as configurações necessárias. Após o primeiro acesso, feito pelo cliente, dados estáticos serão armazenados como mostrado na Figura 4.5.

A compressão de dados reduz significativamente o tamanho dos arquivos, como mostrado na Figura 4.7. Neste caso, houve uma redução

em mais de 40% (quarenta por cento) no tamanho do arquivo. Tal melhoria se torna importante em casos onde há limite de tráfego, pois reduz a quantidade de dados transportados. A melhoria será ajustada no procedimento `configs_apache`.

```
$ ls -l _header.html.erb
-rw-r--r--@ 1 user  user  788 21 Fev 20:32 _header.html.erb
$ cat < _header.html.erb | wc -c
     788
$ gzip < _header.html.erb | wc -c
     326
```

Figura 4.7: Exemplo de utilização do comando *gzip*.

Estas regras têm pesos elevados, pois são a terceira e quarta regras na hierarquia para melhoria de performance direcionada no lado cliente (*SOUDERS, 2007)*. O procedimento responsável por estas regras pode ser verificado na Figura 4.8.

```
 8  def configs_apache
 9    if File.exists?("#{Rails.root}/public/.htaccess")  then
10      File.open("#{Rails.root}/public/.htaccess", "a") do |file_apache|
11        file_apache.puts "<IfModule mod_deflate.c>\n\t
12        AddOutputFilterByType DEFLATE text/html text/plain
13        text/xml text/css application/x-javascript\n</IfModule>\n\n"
14        file_apache.puts "ExpiresActive On\n<FilesMatch '
15        \\.(ico|jpg|jpeg|png|gif|js|css)'>\n\tExpiresDefault
16        'access plus 1 month'\n</FilesMatch>"
17        file_apache.puts "<Directory '#{Rails.root}/public/assets'>
18        \n\tExpiresDefault 'access plus 1 month'\n</Directory>"
19      end
```

Figura 4.8: Procedimento responsável por criar cache e compressão de arquivos.

3 – Separação de CSS e *JavaScript*

Separar dentro da *view* estes tipos de arquivo possibilita, em muitos casos, uma melhoria na performance no carregamento da página, pois os arquivos do tipo *javascript* além de serem baixados, devem ser processados e enquanto o *browser* estiver baixando o *javascript,* as

requisições são bloqueadas até que o arquivo em questão seja baixado e processado. Logo isto demanda tempo e pode acarretar numa espera maior para o carregamento da página, já que por padrão os *browsers* fazem *downloads* de dois arquivos em paralelo (SOURDERS, 2007).

Porém quando há a necessidade de validação utilizando arquivos *javascript,* este procedimento não poderá ser utilizado. Existem duas regras que ditam que estes arquivos devam ser separados onde o tipo CSS devam ser colocados acima dos arquivos *JavaScript (SOUDERS, 2007).* Este regra foi implementada dentro do procedimento `join_jscss`, que está apresentado na Figura 4.9.

```
51 if line =~ /<\/title>/
52   temp.puts "\t\t\t<title><%= title %><\/title>
53   \n\t\t\t<%= csrf_meta_tag %>\n\t\t\t
54   <%= include_stylesheets :workspace, :media => 'all' %>
55   \n\t\t\t<%= yield stylesheets %>\n"
56 else
57   temp.puts "#{line}"
58 end
59 temp.puts "\n\t\t<%= include_javascripts :workspace %>
60 \n\t\t<%= yield javascripts %>\n" if line =~ /<\/body>/
```

Figura 4.9: Código separa os arquivos *javascript* e CSS.

4.2 - MELHORIAS DE PERFORMANCE NO SERVIDOR

Atualmente grandes sistemas que possuem uma quantidade enorme de dados trafegando exigem por parte de seus desenvolvedores uma atenção especial para que a escalabilidade e performance de suas aplicações melhorem e que exista uma boa iteração entre seus servidores e as máquinas clientes. Diante de milhares de usuários cadastrados, o grande número de requisições pode ocasionar gargalos e com isto a perda de performance do sistema. O *framework* disponibiliza ao desenvolvedor saídas para tratar problemas que podem comprometer a boa utilização do sistema. Dentro da gema proposta serão introduzidas três melhorias de performance no lado do servidor que são:

1 – Configuração de um *host* para arquivos estáticos

A criação de um servidor exclusivo para *download* de arquivos estáticos desonera o servidor principal da tarefa de enviar a máquina cliente arquivos estáticos que não exigem nenhum processamento adicional. Com isso o servidor principal poderá direcionar o processamento para tarefas mais importantes e trabalhosas. A configuração dentro do *Rails* é fácil e será feita pelo procedimento `config_static_files`, visto na Figura 4.10. Após a alocação do código será necessário modificar nos *links* dos arquivos que serão baixados o nome do subdomínio do servidor, para `assets%d` onde o símbolo %d será substituído pelo intervalo entre 0(zero) e 3(três).

```
106  file.puts("\n\tconfig.action_controller.asset_host = 'http:\/\/assets%d.#{server}'\n")
```

Figura 4.10: Procedimento `config_static_files`.

2 – *Caching* no lado servidor

Existem três formas de utilizar *caching* no *Rails.* E são elas: *Action Caching* que faz o armazenamento de uma determinada *action* dentro no disco*, Page Caching* que armazena alguma página, preferencialmente, aquela de acesso público e *Fragment Caching* armazena pedaços de alguma visualização para o disco. Dentro da gema proposta quem escolherá qual controlador e página será cacheada é o usuário. O procedimento `cache_server_side` é o responsável por esta operação de criar um *cache* apenas para ações e páginas informados pelo usuário, não estendendo-se a fragmentos de página. O procedimento, ilustrado na Figura 4.11, cria um observador que fará a operação de expiração do *cache* sempre que o usuário executar operações de inserção, atualização ou deleção dentro do banco de dados. Assim o conteúdo da página será atualizado e um novo *cache* efetuado. O controlador e a ação serão

escolhidos pelo usuário, através de parâmetros passados na execução da gema, assim os mesmos serão monitorados pelo observador que ficará responsável por administrar todas as mudanças ocorridas.

```
81  ob.puts "class #{controller.capitalize}Sweeper < ActionController::Caching::Sweeper"
82  ob.puts "\tobserve #{controller.capitalize!}"
83  ob.puts "\tdef expire_cached_content(#{controller.downcase})\t"
84  ob.puts "expire_page :controller => '#{controller.pluralize}', :action => '#{action.downcase}'"
85  ob.puts "\texpire_fragment(%r{#{controller.pluralize}/.*})\nend "
86  ob.puts "alias_method :after_save, :expire_cached_content"
87  ob.puts "alias_method :after_destroy, :expire_cached_content\nend"
```

Figura 4.11: Procedimento criador do cache para controlador e ação passados por parâmetros.

3 – Memória

Uma preocupação dos administradores de grandes sistemas é o consumo de memória que sua aplicação pode gerar no servidor. Tal problema deve desprender a máxima atenção do administrador, pois o alto consumo de memória pode acarretar num mau funcionamento ou até mesmo na inutilização de seu servidor. Melhorar a gerência da memória do servidor da aplicação que a execute foi o foco. Será utilizada a gema *memcached* com intuito de administrar o consumo de memória do servidor.

Dentre as opções o administrador poderá executar o balanceamento do consumo de memória no seu servidor ou em múltiplos servidores.

```
169 require_memory
170 File.open("#{app_dir}","r") do |file_controller|
171   file_controller.readlines().each do |line|
172     if line =~ /class ApplicationController < ActionController::Base/
173       temp1.puts "class ApplicationController < ActionController::Base\n\t"
174       temp1.puts "session :cache => MemCache.new('#{s1}:11211', '#{s2}:11211')\n"
175     else
176       temp1.puts "#{line}"
177     end
178   end
179 end
```

Figura 4.12: Corpo do principal do procedimento *multi_memory*.

Por padrão o servidor local na porta 11211 será configurado, utilizando 512MB de memória RAM. Para a configuração de um ou dois servidores externos. O administrador deverá informar na forma de parâmetros os *ip's* dos servidores e utilizar o procedimento `multi_memory,` presente na Figura 4.12, que receberá até 2 servidores para o balanceamento externo de memória.

4.3 – ANÁLISE DOS RESULTADOS

Nesta seção as melhorias propostas serão analisadas e seus resultados apresentados, bem como todos os componentes, *hardware* e *software,* utilizados nos testes. Esta seção tem por objetivo comprovar ou não se os objetivos propostos foram atingidos.

4.3.1 – *Hardware*

Para a medição do ganho de performance após a execução da gema *o hardware* utilizado foi:

- Processador: Pentium Core 2 Duo, 2,16 GHz.
- Memória: 1 GB, 667 MHz DDR2 SDRAM.
- Placa Mãe: Intel ICH7-M AHCI.

4.3.2 – S*oftware*

Está subseção informa quais os *softwares* utilizados. O sistema operacional, bem como programas de apoio utilizados nos testes feitos.

- Sistema Operacional: Mac OS X, Versão 10.6.7.
- Browser: Safari e Google Chrome.
- IDE: TextMate, Versão 1.5.10.

4.3.3 – Discussão das melhorias

A junção e redução do número de arquivos do tipo *javascript* e CSS apresentaram um alto ganho de performance. O número de requisições para cada tipo de arquivo ficou constante em apenas uma requisição. Os resultados do tamanho dos arquivos antes e após da utilização da gema de apoio *jammit* podem ser verificados na Tabela 4.1.

Qtd	Antes do *Jammit (Bytes)*	Após *Jammit (Bytes)*	Ganho %
1	211978	94356	55,48
3	316032	190749	39,64
5	661569	370260	44,03
7	702206	394805	43,77
11	772099	438772	43,17

Tabela 4.1: Resultados da utilização da gema *jammit.*

A utilização de páginas cacheadas é sempre mais performática do que páginas que não utilizam está técnica, já que após o primeiro carregamento, os arquivos ficam cacheados tanto na máquina cliente quanto na máquina servidora, diminuindo no caso do cliente o número de requisições e no servidor retirando a necessidade de busca em disco o que demanda mais tempo (SOUDERS, 2007). “Quando todos os componentes do documento são lidos a partir do cache do navegador, o tempo de resposta é diminuído em até 50% ou mais.” (SOUDERS, 2007).

A utilização da compressão utilizando o *gzip* do servidor Apache obteve um ganho de performance na redução do arquivo, onde através de testes práticos a diminuição foi em média de 40% no seu tamanho.

As melhorias feitas no gerenciamento e balanceamento de memória nas aplicações *Ruby on Rails*, no servidor, poderá aumentar a capacidade total de processamento de requisições e diminuir o gargalo de utilização de memória até o limite físico das máquinas auxiliares.

A configuração de um servidor estático no mínimo dobrará o número de *downloads* em paralelo feitos pelo cliente, o que configura uma melhora de performance significativa na aplicação de acordo com a equação 4.1, levando em conta que os *browsers* testados executam o *download* de dois elementos em paralelo:

$$Total = NumS \times 2 \qquad \textit{(eq. 4.1)}$$

onde:

Total = Número total de arquivos baixados em paralelo.

NumS = Número de servidores destinados para arquivos estáticos.

4.4 – CONCLUSÃO

Com a abordagem das regras propostas por *Steve Souders* espera-se que qualquer aplicação que execute a gema tenha um ganho de performance no lado cliente, já que grande parte das regras mais importantes, propostas por *Steve Souders*, foram tratados dentro de procedimentos na gema. Algumas delas, como redução do número de requisições, minimização dos arquivos CSS e *javascript*, compressão de dados utilizando o *gzip,* demonstraram serem capazes de melhorar a performance em até 40%. Dentro de um grande sistema que possua uma alta taxa de tráfego, com milhares de usuários, isto pode significar uma melhora relevante em sua performance.

Os procedimentos que visam a melhoria de performance no servidor utilizam técnicas que são largamente utilizadas na internet. O aumento da capacidade da memória através da utilização de servidores auxiliares que farão o balanceamento da mesma, a criação do *cache* no servidor ocasiona uma diminuição no tempo de busca por arquivos já que a operação de busca em *cache* é mais performática do que a em disco e a utilização de um servidor exclusivo para arquivos estáticos diminui o

processamento por parte do servidor e aumenta a capacidade de *download* de arquivos em paralelo pelo cliente. Com a utilização de tais melhorias uma aplicação poderá obter um ganho de performance e uma gerência melhor no tratamento de pontos críticos como o consumo de memória.

5 – CONCLUSÃO

A gema foi desenvolvida na linguagem de programação *Ruby*, com o objetivo principal de auxiliar administradores de sistema a melhorar as questões performáticas em suas aplicações *Ruby on Rails,* utilizando soluções propostas por autores, como *Steve Souders*, e técnicas largamente utilizadas na Internet, como o *cache* que comprovadamente melhoram a performance em aplicações que a utilizam.

A gema está disponibilizada como arquivo *gem* no repositório do site *Github* (https://github.com/eduardo82/performance.gem) e poderá ser baixada como qualquer outra *gem* utilizada pelo *Ruby on Rails.* No desenvolvimento do trabalho notou-se que a criação de tarefas isoladas para que o administrador escolha apenas abordar determinada área de sua aplicação aumenta a flexibilidade da gema e proporciona maior liberdade no momento de sua execução.

Através de testes práticos realizados, pesquisas em livros e Internet e análise de resultados ficou comprovado que as técnicas abordadas pela gema para melhoria de performance em aplicações *Ruby on Rails* alcançaram o objetivo de: obtenção de ganhos, maximização da iteração cliente-servidor e também minimização de pontos críticos, com o aumento e balanceamento de memória.

As maiores dificuldades encontradas no desenvolvimento foram a escolha de técnicas mais relevantes e necessárias levando em consideração seu grau de impacto na melhoria de performance para que seja minimizado ao máximo questões que possam comprometer ou diminuir funcionalidades da aplicação.

Para que a gema funcione não será necessária nenhuma configuração adicional dentro da aplicação, suas dependências serão instaladas automaticamente no momento de sua execução o que proporciona mais agilidade a gema.

Por ser um projeto *Open Source* qualquer programador ou administrador de sistema que queira introduzir ou modificar os procedimentos dentro da gema com o objetivo de aumentar sua abrangência, poderão fazê-lo através de *commits* no repositório em que ela está hospedada. Com isso inúmeras possibilidades no que tange a trabalhos futuros poderão ser executados dentro da gema, como por exemplo: melhorias na interação com o banco de dados, procurando por soluções mais rápidas, no que tange a consultas e menor consumo de memória, no *ActiveRecord.*

REFERÊNCIAS BIBLIOGRÁFICAS

AKITA, F. **Repensando a web com Rails.** 1ª Edição, Rio de Janeiro: Brasport, 2006. 508p.

BATES, R. **New Gem with Bundler.** 2010. Disponível em: <http://www.railscasts.com/245_new_gem_with_bundler.mov>. Acesso em: 8 jan. 2011, 20:43hs.

COLLINGBOURNE, H. **The little book of Ruby.** 2ª Edição, EUA:Dark Neon, 2008. 87p.

EDIGER, B. **Advanced Rails.** 1ª Edição, EUA:O'Reilly, 2007. 355p.

FERNANDEZ, O. **The Rails 3 Way.** 1ª Edição, EUA:Pearson, 2010. 760p.

FULTON, H. **The Ruby Way: Solutions and Techniques in Ruby Programming.** 2ª Edição, EUA:Addilson Wesley Professional, 2006. 888p.

IBM. **Building Object-Oriented Frameworks.** EUA, 1997. 23p. Disponível em: <http://www.ibm.com/java/education/oobuilding/>. Acesso em: 08/10/2010.

NETO, O. **Entendendo e Dominando o Java para Internet.** 1ª Edição, São Paulo:DigeratiBooks, 2006. 320p.

NETO, O. **Entendendo e Dominando o Java.** 2ª Edição, São Paulo:DigeratiBooks, 2007. 416p.

PORTAL DE GOVERNO ELETRÔNICO DO BRASIL. Pesquisa aponta crescimento do acesso à internet nos lares brasileiros. **Notícias**. Portal de Governo Eletronônico do Brasil. Publicado em 06/04/2010. Disponível em: <https://www.governoeletronico.gov.br/noticias-e-eventos/noticias/pesquisa-aponta-crescimento-do-acesso-a-internet-nos-lares-brasileiros>. Acesso em: 10/09/2010.

SMETANA, G. **IPv6.** São Paulo. 20p. Disponível em: <http://www.teses.usp.br/teses/disponiveis/3/3141/tde-10032004-002135/publico/mestradoGeorgeMarcelSmetana.pdf>. Acesso em 23/10/2010.

SOUDERS, S. **Even Faster Web Sites.** 1ª Edição, EUA:O'REILLY, 2009. 256p.

SOUDERS, S. **High Performance Web Sites. Essential Knowledge for Frontend Engineers.** 1ª Edição, EUA:O'REILLY, 2007. 170p.

TATE, B. **Ruby on Rails Executando.** 1ª Edição, Rio de Janeiro:ALTA BOOKS, 2006. 174p