



**INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
FLUMINENSE**
Campus Campos-Centro

Secretaria de Educação
Profissional e Tecnológica

Ministério
da Educação



BACHAREL EM SISTEMAS DE INFORMAÇÃO

ALEXANDRE DOS SANTOS SAMPAIO SILVA

LAÍS STELLET DA SILVA

TÍTULO

Campos dos Goytacazes/RJ
2014



**INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
FLUMINENSE**
Campus Campos-Centro

Secretaria de Educação
Profissional e Tecnológica

Ministério
da Educação



BACHAREL EM SISTEMAS DE INFORMAÇÃO

ALEXANDRE DOS SANTOS SAMPAIO SILVA

LAÍS STELLET DA SILVA

TÍTULO

Trabalho de conclusão de curso apresentado ao Instituto Federal Fluminense como requisito obrigatório para obtenção de grau em Bacharel de Sistemas de Informação.

Orientador: Prof. DSC Maurício José Viana de Amorim

Campos dos Goytacazes/RJ

2014

ALEXANDRE DOS SANTOS SAMPAIO SILVA

LAÍS STELLET DA SILVA

Tema – ???

Trabalho de conclusão de curso apresentado ao Instituto Federal Fluminense como requisito obrigatório para obtenção de grau em Bacharel de Sistemas de Informação.

Aprovada em ?? de ?? de 20??

Banca avaliadora:

Prof. DSC Maurício José Viana Amorim
Instituto Federal de Educação, Ciência e Tecnologia Fluminense / Campus Campos
Centro

???

??.

AGRADECIMENTOS

????????????

O computador não é mais o centro do
mundo digital.

Tim Cook

RESUMO

Este trabalho descreve sobre o desenvolvimento de um sistema de controle de atividades complementares imposto pelo Instituto Federal Fluminense, criado sob a linguagem de programação Ruby com o auxílio de seu web framework o Rails com intuito de realizar um desenvolvimento ágil com auxílio de Behavior Driven Development. Para concretizar esse trabalho durante todo o processo de desenvolvimento e de test foram utilizado varios conceitos de Ruby. Sendo o objetivo desta ferramenta é auxiliar tanto o aluno bem como um professor avaliador a computar notas para estas atividades. Foi-se utilizado um sistema a base UNIX, e um ambiente de desenvolvimento para criar a aplicação, bem como base de dados para se criar todo o procedimento. Foram utilizadas as bases Mysql e MongoDB com comprativo entre bancos de dados relacionais e não relacionais.

PALAVRAS-CHAVE: Serviço Web, Software livre, UNIX, Ruby, Rails, Mysql, NoSQL, MongoDB

ABSTRACT

This work describe about the development of an app from zero to deploy call System for a complementary activities control created by the computing coordinating from the Fluminense Federal Institute. This app was created over the programming language called Ruby with the assistance of his own web framework called Rails with the intention to perform an agile web development with the aid of Behavior Driven Development. To accomplish this work throughout the development process and test process, various concepts of Ruby were used. Since the purpose of this tool is to assist both the student as well as a reviewer to compute scores for these activities. A system based on UNIX were used, and a development environment to create the application and database to create the entire procedure. The MySQL and MongoDB databases were used to comprativo between relational databases and non-relational.

KEYWORDS: Serviço Web, Software livre, UNIX, Ruby, Rails, Mysql, NoSQL, MongoDB

LISTA DE FIGURAS

2.1	Modelo MVC básico	18
2.2	Modelo MVC hibrido	18
2.3	Modelo MVC Client-Side	19
2.4	Modelo MVC	19
2.5	Estrutura básica de arquivos e diretórios	24
2.6	Parte dos arquivos criados após o scaffold.	25
3.1	Diagrama do Teorema de CAP(Brewer)	29

SUMÁRIO

1	INTRODUÇÃO	11
1.1	Objetivos	11
1.2	Justificativa	11
1.3	Organização do Trabalho	11
2	TECNOLOGIAS EMPREGADAS	12
2.1	RVM (Ruby Version Manager)	12
2.1.1	Ruby	13
2.1.2	RubyGems	15
2.1.3	Como Ruby carrega RubyGems	15
2.1.4	Versionamento de Gems	16
2.2	Organização de aplicações web	17
2.2.1	Aplicações ricas em client-side	17
2.3	Ruby On rails	20
2.3.1	DRY	20
2.3.2	CoC	21
2.3.3	Active Record	21
2.3.4	Action Controller	22
2.3.5	Action View	22
2.3.6	Action Support	22
2.3.7	Action Mailer	23
2.3.8	Rake	23
2.3.9	Migrations	23
2.3.10	Organização de arquivos em Rails	23
2.3.11	Scaffolding	25
2.3.12	Comunidade Rails	25
2.4	Git	26
3	Banco de Dados - SGBDs x NoSQL	27

3.1	Breve Histórico	27
3.2	SGBDs Relacionais	28
3.3	Teorema de CAP	29
3.3.1	C – Consistência	29
3.3.2	A – Disponibilidade	30
3.3.3	P – Tolerância ao Particionamento	30
3.4	NoSQL	30
3.4.1	Escalonamento horizontal	31
3.4.2	NoSQL 1: Sistemas CP	32
3.4.3	NoSQL 2: Sistemas AP	32
3.5	Comparativo entre SGBDS Relacionais e Não Relacionais	32
4	SGBDS	33
4.1	Porque Utilizar?	33
4.2	Banco de Dados Relacionais	33
4.3	NoSQL	33
4.4	Comparativo entre SGBDS Relacionais e Não Relacionais	33
	REFERÊNCIAS BIBLIOGRÁFICAS	34

1 INTRODUÇÃO

1.1 Objetivos

1.2 Justificativa

1.3 Organização do Trabalho

2 TECNOLOGIAS EMPREGADAS

Para o desenvolvimento do Atividades Complementares foi essencial o uso de algumas ferramentas as quais pudessem ser incorporadas e tornassem possível a criação de funcionalidades para este projeto.

Ao longo de muitos anos, uma pergunta é feita por vários desenvolvedores. Qual o melhor Framework de desenvolvimento web já criado? Contudo esta é uma pergunta respondida de diversas formas, com base em vários argumentos e situações. E a resposta entretanto continua a mesma - Aquela que aumenta a produtividade e diminui esforços no desenvolvimento. E depois de muito estudo e debate fora concluído que um Framework que reduza códigos gigantes e por muitas vezes difíceis de se ler e que sem margem de duvida diminuam o aprendizado são aqueles que aumentam a produtividade. Se buscarmos mais a fundo iremos nos debater com vários web Framework que focam em produzir aplicações, sejam elas web ou não. No entanto essas pecam em dois pontos vitais: produtividade e aprendizado. Um equilíbrio teve de ser estabelecido entre esses fatores e foi nesse momento que Ruby surgiu precedido do Rails como seu Framework de desenvolvimento web.

Serão apresentadas as seguintes tecnologias: RVM como ambiente virtual para o desenvolvimento, Ruby como linguagem de programação, Rails como web framework, Mysql como base de dados inicial, MongoDB como base de dados conclusiva e o mais importante RubyGems para auxiliar na produção do sistema, git como ferramenta de repositório de código compartilhado com membros do projeto.

2.1 RVM (Ruby Version Manager)

RVM é uma ferramenta de linha de comando que permite ao usuário facilmente instalar, gerenciar e possuir varios ambiente de desenvolvimento ruby em uma só máquina com suas respectivas versões de rubygems.

2.1.1 Ruby

Foi desenvolvida em 1994, e apresentada ao publico em 1995 por **Yukihiro Matsumoto**, que é mundialmente conhecido como Matz que para criar ruby uniu partes das suas linguagens favoritas (*Perl, Smalltalk, Eiffel, Ada, e Lisp*) para formar uma nova linguagem que equilibra a *programação funcional com a programação imperativa*. Passou a se tornar realmente reconhecida através de **Dave Thomas** , mais conhecido como um dos “*Programadores pragmáticos*”, que adotou o Ruby como uma de suas linguagens preferenciais e acabou escrevendo um dos livros mais completos sobre a linguagem, o Programming Ruby. Com isso o número de adeptos a essa linguagem subiu muito rápido no ocidente. Ruby é uma das únicas linguagens nascidas fora do eixo EUA/Europa, sendo criada no Japão.

Matz criou uma linguagem mais poderosa que Perl e mais orientada a objetos que python. Em Ruby tudo é objeto, e possui métodos que podem ser facilmente acessados e modificados. O tornando assim uma linguagem mais simples de se ler e ser compreendida, facilitando o desenvolvimento e manutenção de sistemas escritos com com essa base. Um dos objetivos principais da linguagem é a praticidade. É possível que seja feito um algoritmo simples, sem precisar que se preocupe com as limitações da linguagem ou do interpretador.

O Ruby possui algumas características para manter a sua praticidade, como, uma linguagem enxuta que quase não há necessidade de colchetes e outros caracteres; a disponibilidade de diversos métodos de geração de código em tempo real, como os "attribute accessors";

"Ruby é simples na aparência, mas é muito mais complexo no interior, tal como o corpo-humano!" - Ruby-Talk, (MATZ, 2000).

Uma forma prática de se instalar bibliotecas em ruby é através de **RubyGems** , com ela é possível instalar e atualizar bibliotecas com uma única linha de comando, de maneira muito similar com os gerenciadores de pacotes de ambientes operacionais linux ou unix; Ruby possui uma notação muito útil aos desenvolvedores, estas são chamadas de blocos de código, que ajudam o programador a passar um ou mais trechos de instruções para o método descrito em suas classes concretas. Ruby possui o contexto de Mixins, que simula herança múltipla, sem cair em seus respectivos problemas encontrados em outras linguagens. Ruby é classificado como dinamicamente tipado, por esse meio todos os tipos são objetos, não há tipos primitivos como era e ainda é feito em outras linguagens do genero como **C++, Java(...)** , bem como suas definições de classes. No entanto variáveis de instância que por sua vez referenciam estes objetos não possuem um tipo específico. Um exemplo clássico em ruby seria uma reescrita da classe Fixnum nativa do ruby.

```

1 [Irb]
2 [irb - prompt]
3   2.1.0 :001 > class Fixnum
4   2.1.0 :002 >   alias_method :old_add, :+
5   2.1.0 :003 >   def plus(other)
6   2.1.0 :004 >     self.old_add(other) * 2
7   2.1.0 :005 >   end
8   2.1.0 :006 > end
9
10 [irb - prompt]
11   2.1.0 :007 > numero = 2 + 2
12   2.1.0 :008 > numero
13   "8"
14   2.1.0 :009 > numero = nil

```

Código 2.1: Exemplo de uma *Reescrita* de Classe

O operador *plus* ou + é um método em Ruby, ao contrário de outras linguagens existentes. O resultado desse método reescrito na classe Fixnum diz que todo e qualquer numero somado com o método soma, irá sempre executar a operação de soma em conjunto com o operador de multiplicação neste caso todo numero somado será multiplicado por dois após a operação anterior, e logo em seguida pelo fato de ruby ser dinamicamente tipado, o valor da variável **numero** é alterado para nil, o que significa que podemos alterar o contexto de uma variável ou até mesmo inserção de código em tempo de execução.

Ruby possui uma ferramenta muito interessante, semelhante ao array visto em outras linguagens o ruby hashes, bastante similar ao dicionário de dados do python. Um ruby hashe utiliza chaves em vez de colchetes precedido de literais. O literal deve fornecer: uma chave e um valor agregados. Por exemplo, se quiséssemos mapear os dados de um usuário poderíamos fazê-lo da seguinte forma:

```

1   user_section = {
2     name: 'xpto' ,
3     email: 'xpto@xpto.com' ,
4     password: 'headwind' ,
5     nickname: 'headwind' ,
6     preferred_language: 'Ruby On Rails with MongoDB'
7   }

```

Código 2.2: Exemplo de um *Ruby Hash*

O conceito acima é muito semelhante a forma como o MongoDB manipula seus documentos um contexto que será explicado mais a frente.

Ultimamente a linguagem tem sido foco da mídia especializada devido ao seu web framework feito em Ruby, o Rails desenvolvido por (HANSSON, 2006). Ainda hoje, toda a responsabilidade, quanto a, implementações de novas funcionalidades, é do Matz. Todas as decisões relacionadas à linguagem tem que passar por ele antes de serem implementadas e virem

à publico. E mesmo assim a comunidade Ruby é forte o suficiente pra sobreviver caso alguma coisa aconteça com o Matz. Pois há muitas pessoas que estão conectadas ao código tanto quanto o próprio Matz. Uma das grandes diferenças das outras tecnologias open-source, é que não tem uma empresa bancando os seus custos. O projeto sobrevive de doações feitas pelos usuários satisfeitos e por empresas que conseguiram aumentar sua produtividade e performance usando apenas Ruby ou Ruby On Rails. Em uma de suas declarações Matz fala sobre o que ele esperava obter quando criou a linguagem:

"Eu conhecia muitas linguagens antes de criar o Ruby, mas nunca estava satisfeito com elas. Elas eram feias, rigorosas, mais complexas ou mais simples do que eu esperava. Eu queria criar a minha própria linguagem que me satisfizesse como programador. Eu sabia muito sobre o público a ser alcançado: eu mesmo. Para minha surpresa, muitos programadores do mundo todo sentiam o mesmo que eu. Eles ficaram felizes quando descobriram e programaram no Ruby. Do começo ao fim do desenvolvimento da linguagem Ruby, concentrei minhas energias para fazer uma programação rápida e fácil. Todas as características do Ruby, incluindo as características de orientação a objetos, são designadas a funcionar com programadores comuns (por exemplo: eu) que esperam que elas funcionem. A maior parte dos programadores acha que ele é elegante, fácil de usar e sentem prazer em usá-lo."(MATZ, 2000).

2.1.2 RubyGems

Uma RubyGem ou simplesmente *Gem* é uma biblioteca como em qualquer outra linguagem de programação já criada, por exemplo: **Python, Java, C++, C**, específicas para Ruby, que fornecem um formato padrão para aplicações. Uma Gem é escrita especialmente para facilitar o uso de determinada funcionalidade. Uma Gem é um conjunto de arquivos feitos em Ruby, etiquetados com nome e versão, cada uma possui, em seu escopo todas as características correspondente a sua arquitetura via um arquivo de configuração chamado "**gemspec**". Tomando como exemplo a gem 'rspec-rails' que possui em seu escopo arquivo rspec-rails.gemspec que possui toda a especificação desta desde qual o grupo responsável por mante-la com atualizações constantes, licenças e dependências. Gems podem ser utilizadas para estender ou modificar certas funcionalidades, geralmente são distribuídas por outros desenvolvedores Ruby mais conhecidos como **rubistas**, várias delas possuem até mesmo comandos específicos auxiliar e agilizar o desenvolvimento, além de que em Ruby rubygems podem ser integradas umas com as outras para facilitar ainda mais os ruby programadores.

2.1.3 Como Ruby carrega RubyGems

Antes de tudo é muito útil saber como o Ruby carrega arquivos de dependência. O Ruby seja na versão 1.9.3 ou mais recente predispõem de um comando bem simples o require "ar-

quivo"com ele é possível carregar arquivo(s) diretamente ou especificando o caminho absoluto. Contudo existe ainda uma outra forma pouco usada o load "arquivo.rb". A única diferença é que com load deve ser colocado a estensão do arquivo e além disso se ao tentar executar o require novamente essa operação será negada pois este carrega uma vez esse arquivo, já o load permite carregar varias vezes. Load é muito útil quando se está testando um arquivo que está sofrendo alterações constantes. Existe ainda uma outra forma de se carregar arquivos é o require File.expand_path("../spec_helper")

```

1 [Irb]
2 [irb - prompt - 1.9.3]
3   1.9.3 :001 > require 'lib/fixnum'
4     true
5   1.9.3 :002 >
6
7 [irb - prompt - 1.9.3 - Full-Path]
8   1.9.3 :001 > File.expand_path("../spec_helper")
9     true
10  1.9.3 :002 >
11
12 [irb - prompt - 2.1.0]
13  2.1.0 :001 > load 'lib/fixnum.rb'
14     true
15  2.1.0 :002 >

```

Código 2.3: Exemplo de require e load Ruby 1.9.3 - 2.1.0

Como pode ser visto o comando require pode carregar arquivos usando caminhos relativos. Porém também pode ser usado caminhos absolutos. Para tal quando se deseja carregar alguma dependência em um projeto Ruby existe um arquivo chamado "**rubygems.rb**" localizado em **./rvm/rubies/ruby-2.1.0/lib/ruby/2.1.0** ao qual contem toda a especificação de Gems como carregalás na versão decorrente do ruby instalado. Logo se precisa carregar esse arquivo toda vez que se deseja utilizar alguma gem:

```

1 require 'rubygems'
2 require 'BCrypt'

```

Código 2.4: Exemplo de require 'rubygems'

2.1.4 Versionamento de Gems

Versionamento é considerado por muitos a parte mais importante das RubyGems. É possível ter diversas versões na mesma máquina carregadas em paralelo, essa é considerada a parte mais vantajosa e a principal fonte de confusão. Por exemplo: ao utilizar o comando 'gem list -local' uma lista será retornada contendo todas as Gems instaladas no momento. Essa lista pode e deve variar de máquina para máquina de acordo com as gems instaladas, para exemplificar esse conceito seria como descrito na saída do terminal a seguir:

```
1 >> mongoid ( 3.0.0 , 4.1.0 )
```

Código 2.5: Exemplo de versionamento de gems

Nesse trecho se carregarmos essa Gem uma mensagem de erro seria exibida, devido ao nome da Gem que não é o nome que o comando 'require' precisa, bem como a versão que se deseja carregar. No entanto como fazer para carregar essa gem se o nome da mesma e o arquivo Ruby diferem nesse contexto?

"Quando a convenção falha precisamos dizer explicitamente o que queremos."(AKITA, 2009).

```
1 >> require 'rubygems'  
2 >> gem 'mongoid'  
3 >> require 'mongoid'
```

Código 2.6: Exemplo de versionamento de gems

O que foi feito no trecho anterior resolve o problema de nomes e carrega a versão mais atual da gem 'mongoid', caso a gem requerida seja uma versão que difere da atual, o desenvolvedor deverá especificar qual a versão da gem propriamente dita quer utilizar:

```
1 require 'rubygems'  
2 gem 'mongoid', '~> 3.0.0 '  
3 require 'mongoid'
```

Código 2.7: Exemplo de versionamento de gems

2.2 Organização de aplicações web

Construir aplicações web em Ruby on Rails é um pouco mais profundo do que simplesmente construir páginas atrás de páginas como era feito antigamente. A razão para tudo isto é que ele é preparado para atender e criar aplicações modernas e arrojadas(sofisticadas) e isto quer dizer que a aplicação desenvolvida sobre esta plataforma não deve apenas responder por páginas HTML mas de forma adequada e dinâmica em aplicações ricas em client-side em conjunto com outros frameworks como backbone.js entre outros.

2.2.1 Aplicações ricas em client-side

Antes de pensarmos em client-side deve-se ter em mente um conceito de aplicações web. Olhando o modelo server-side - MVC tradicional - Model, View e Controller, sendo todos executados no lado do servidor, gerando todo o html em conjunto com suas respectivas actions(requisições), para serem visualizadas pelo usuário em um web-browser

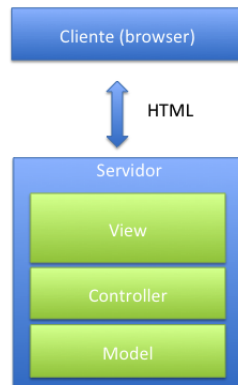


Figura 2.1: Modelo MVC básico

Foi se criado um formato um pouco mais rebuscado, mais evoluído e nesse formato tem-se aplicações onde o html é gerado no lado do servidor, mas foi criada uma técnica conhecida por muitos como ajax, para atualizar partes de páginas ou até mesmo funcionalidades implementadas para melhorar o desempenho do usuário evitando um reload completos da página a cada iteração. Contudo ainda neste ponto parte do MVC é executado no lado do servidor e apenas a parte das Views são executadas no lado do cliente. Este modo eram chamado de híbrido por muitos.

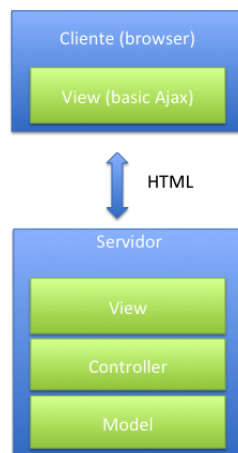


Figura 2.2: Modelo MVC híbrido

Depois de muito se estudar o conceito de modelo MVC - Standard e MVC - hybrid um terceiro modelo surgiu e é nesse que as aplicações client-side executam atualmente e neste a arquitetura MVC é toda executada em client-side (Lado do cliente). O Model passa ter entre tanto duas atribuições de responsabilidade, a de fornecer toda a API bem como as validações executadas no lado do servidor, e outra que é executada no lado do cliente que também oferece validações sobre a estrutura do modelo exibido.

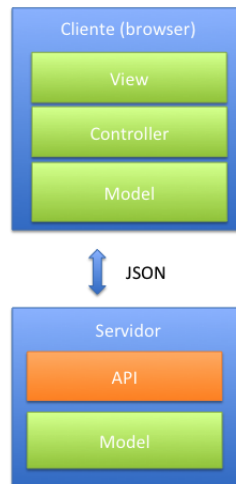


Figura 2.3: Modelo MVC Client-Side

Em resumo o Modelo MVC pode ser descrito como uma série de requisições solicitadas pelo Browser e interpretadas pelo servidor que possui uma via de mão dupla.

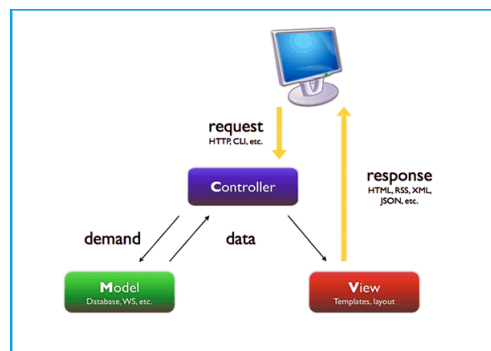


Figura 2.4: Modelo MVC

Desenvolver aplicações voltadas para client-side possuem algumas vantagens importantes como:

- **Melhor desempenho para o usuário:** sendo esta uma das principais vantagens no que diz respeito ao desempenho de cada browser, não ter que precisar fazer requisições a todo momento, na hora de fazer dowload, ou atualizar a página completa a cada interação do usuario com esta. Aplicações feitas em client-side funcionam tao bem quanto aplicações feitas para desktop.
- **Melhor desempenho na transferência de dados:** Há também um ganho considerável na taxa de transmissão dos dados, ao invés de carregar o cenário da página html completa a cada interação do usuário, na arquitetura client-side todo o cenário é transferido na primeira solitação e as requisições seguintes são responsáveis por trafegar apenas os dados consolidados entre o cliente e o servidor, normalmente no formato JSON.

- **Facilidade de manutenção:** com aplicações em client-side a API fornecida pelo servidor possibilita a facilidade de manutenção de forma independente ao usuário, de tal forma que o cliente não perceberá quando a página sofrer atualizações em tempo não real.
- **Redução de carga no lado do Servidor:** O sistema completo passa a ser enviado ao usuário final através de arquivos html, css e javascript que podem ser comprimidos e distribuídos através de CDN's com facilidade. Uma vez baixados esses arquivos são mantidos em cache no browser da preferência do usuário. O servidor tem a responsabilidade apenas de fornecer uma API, enviar e receber os dados no formato JSON. Dessa forma todo o processamento é responsável por particionar dos dados e a geração de templates fica exclusivamente no lado cliente e não mais no servidor, liberando recursos.

2.3 Ruby On rails

RoR é um web framework escrito sob a linguagem Ruby. Criado por (HANSSON, 2006) em 2003, baseado em seu trabalho no Basecamp, que é uma ferramenta de gerenciamento de projetos 37 signals. No entanto David só lançou o RoR como código aberto em julho de 2004, e mais tarde em dezembro de 2005 foi lançada a primeira versão do Ruby on Rails.

Assim como outros frameworks de aplicação web o RoR utiliza a arquitetura MVC(model view controller), fornecendo um isolamento entre a lógica de negócio dos modelos, a interface com o usuário através das views e a manipulação de todas as requisições no servidor de aplicação. O que contribui muito para que a manutenção do código seja bem mais fácil e flexível.

Ruby on Rails possui uma filosofia que segue dois princípios:

- DRY
- CoC

2.3.1 DRY

Don't Repeat Yourself(Não se repita). Se aplicado corretamente, possibilita a reduzir a duplicação de tarefas dentro de um projeto. Réplicas ou duplicatas de qualquer tipo, dentro de uma aplicação, leva a dificuldade de modificação e manutenção e inconsistência, sem levar em conta em alguns casos a ilegibilidade do source-code. Em RoR, se pode ver este princípio em ação em quase tudo, desde as componentes reutilizáveis em forma de plug-ins para a forma como as tabelas da base de dados escolhida são mapeadas.

2.3.2 CoC

Convenção sobre Configuração ou *programação por convenção* vem do termo em inglês (**Convention over Configuration - CoC**), uma prática de desenvolvimento de software que visa diminuir o número obsoleto de decisões que os desenvolvedores precisam tomar ao longo de seus projetos. Estabelecendo simplicidade sem perder flexibilidade. Quando um desenvolvedor seja ele experiente ou não for iniciar atividades em Rails, o usuário estará sempre na maior parte do tempo interagindo com os controllers, views e models entre outras palavras a arquitetura MVC amplamente vista em design patterns e além desse fator importante estará diretamente conectado para a base de dados escolhida seja ela relacional ou não relacional como no caso dos NoSQL. De tal forma a reduzir a necessidade de configuração pesada.

RoR permite a criação de regras personalizadas, contudo é sempre uma boa ideia usar as convenções que o próprio Rails oferece, essas convenções deverão acelerar o desenvolvimento, manter um código limpo, conciso e legível e o mais importante estas convenções permitem uma navegação muito mais fácil dentro da aplicação. Rails não foi baseado em um único padrão de desenvolvimento, mas sim uma série de padrões. Outros frameworks que faziam parte do núcleo do Rails antigamente foram removidos desse núcleo afim de reduzir o acoplamento e com isso e permitir que quem o esteja utilizando os substituam sem muita dificuldade, mas continuam funcionando e sendo usados em conjunto. Aqui estão alguns deles:

2.3.3 Active Record

Para se entender este item sendo este um dos mais importantes para se construir uma aplicação em Rails é necessário compreender um dos fundamentos mais criteriosos em orientação a objetos o conceito de ORM(Object-Relational Mapping) que pode ser traduzido como Mapeamento Objeto Relacional e segundo (TECTARGET, 2008), trata-se de uma forma rápida e prática de relacionar e endereçar, e manipular objetos sem que seja, necessário se preocupar com a forma ao qual estes se comunicam e se relacionam entre si. ORM possibilita aos desenvolvedores experientes ou novatos à manter a uma perspectiva consistente dos objetos no percurso do projeto, mesmo que haja alterações no código ou até mesmo na base de dados em questão.

De acordo com (BAKHARIA, 2007), o **Active Record** cria uma abstração de *OODB* - (Orientação a Objeto em Banco de dados) onde o rails cria um mapeamento relacional entre tabelas e classes do modelo ao qual estas pertencem. Sendo que o Active Record também fornece uma série de métodos nas próprias classes para trabalhar com a manipulação de dados, como por exemplo criar, salvar, atualizar, deletar, entre outras palavras todas operações para gerenciar o CRUD do modelo descrito. Ao contrário de outras bibliotecas complexas o Active Record não necessita de configurações desse nível, além de ser capaz de se dispor da capacidade

de propor mapeamentos objeto relacional com base em convenções de nomenclatura de tabelas e nome dos campos o que ajuda a justificar a importância do conceito chave de desenvolvimento a idéia de *Convention Over Configuration*(*Convenção Preferível à Configuração*) o que a mesma afirma que com esse pretexto o Active record torna o Ruby On Rails uma das ferramentas de desenvolvimento web mais ágil para a produção de sistemas com banco de dados.

Na Prática todo e qualquer modelo criado pelo gerador de componentes do rails o mesmo estende a classe nativa **ActiveRecord::Base** responsável por abstrair uma entidade contida na base de dados, assim como cada objeto representa uma linha do banco de dados como menciona (RUBY, 2014).

2.3.4 Action Controller

Segundo (ORSINI, 2007), em qualquer aplicação feita em rails, um controller é uma especialização da classe **Action Controller** que oferece uma série de conjunto de regras de negócio para a aplicação Geralmente cada controller responde ao modelo ao qual encontra-se alocado, podendo este ser o que muitos chamam de **Controller Virtual**. Um Controller Virtual também é chamado de controller genérico, que não necessariamente está associado à um modelo.

2.3.5 Action View

Segundo (BAKHARIA, 2007), Action View é um dos componentes responsável apenas por gerar a interação entre a aplicação e o usuário. Um Controller pode ter várias actions(métodos), e para cada action dependendo da situação uma view para esta que automaticamente é reconhecida e exibida. Um action view também é conhecida como template. Esse template pode reproduzir vários elementos, não só o html, mas também outros tipos como XML, JavaScript, PDF. O template nada mais é do que uma renderização de código ruby com Html e possivelmente Javascript.

2.3.6 Action Support

"Active Support é composto por uma série de bibliotecas compartilhadas por todos os componentes gerenciados pelo Rails. Muito do que está acoplado nesse módulo está destinado a uso interno do Rails. No entanto, Active Support também estende algumas das classes internas de Ruby de maneira útil e interessante."(RUBY, 2014).

2.3.7 Action Mailer

ActionMailer é um componente nativo do rails que permite que a aplicação desenvolvida possa enviar correio-eletrônico.

"É importante um sistema que facilite a operação de envio de mensagens de e-mails porque essa é uma operação tem diversas aplicações comuns como: Enviar e-mails de confirmação de cadastro; Notificações de erros ao administrador do site; Confirmação de compra de produtos em lojas virtuais; Newsletters".(BAKHARIA, 2007)

A classe Mailer do Rails possui métodos para diferentes formas de enviar mensagens de acordo com o que a aplicação necessita. O formato de saída dessa mensagem é descrito em sua ActionView de forma muito similar com as actions views gerenciadas pelos erb.

2.3.8 Rake

Rake é conhecido como Ruby Make, um utilitário autônomo Ruby que substitui o utilitário Unix 'make' e usa um "Rakefile" e arquivos com extensão .rake para criar uma lista de tarefas. No Rails, Rake é usado para tarefas comuns de administração, especialmente os mais sofisticados que constroem fora de si. Pode-se obter uma lista de tarefas Rake disponíveis simplesmente digitando rake - tasks. Cada tarefa tem uma descrição, o que deve ajudar a encontrar funções específicas para o que se deseja

2.3.9 Migrations

O Migration pertence alterar o esquema de banco de dados ao longo do tempo de uma forma consistente e fácil. Ele utiliza uma DSL do ruby para que não tenha que escrever SQL a mão, fazendo assim com que as mudanças no banco de dados seja independente. Cada migration é uma nova "versão" do banco de dados. Um esquema começa vazio e a cada migration ele é modificado para acrescentar ou remover tabelas, colunas ou entradas necessárias.

2.3.10 Organização de arquivos em Rails

RoR se dispõe de ferramentas de desenvolvimento rápido e sem muito esforço. Para se iniciar uma nova aplicação em rails bastaria usar o comando inicial:

```
>> rails new my_new_rails_app [ -d ] { options }
```

Código 2.8: Exemplo de uso de scaffold

Ao utilizar esse comando inicial toda uma estrutura de arquivos e diretórios é criada.

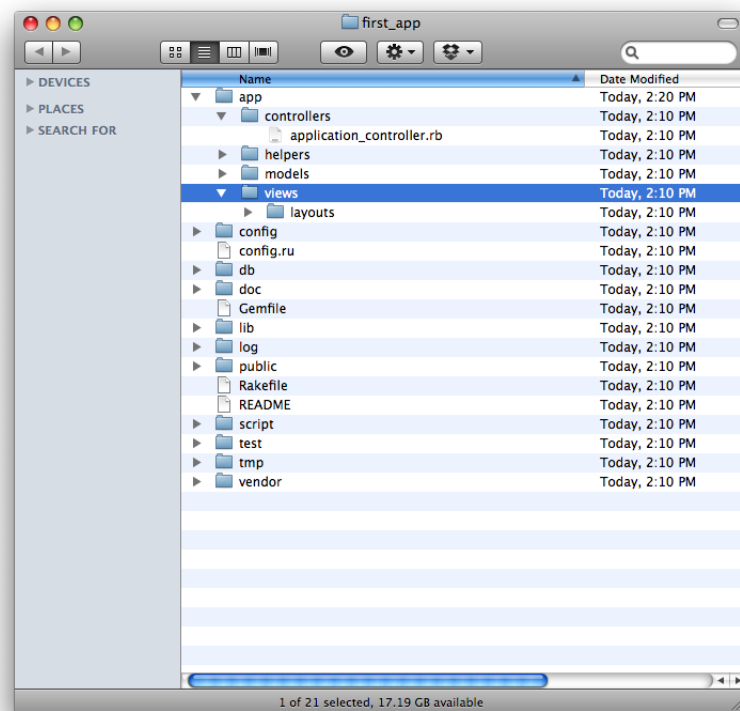


Figura 2.5: Estrutura básica de arquivos e diretórios

É possível ainda passar parâmetros para a criação da mesma tais como: o `-d` significa qual a base de dados a ser utilizada, RoR possui três ambientes de integração: **development** responsável apenas pela etapa de desenvolvimento, este recarrega todas as classes sempre que uma nova action é requisitada, portanto uma nova cópia da classe é obtida, incluindo qualquer alteração recente na mesma, **test** - o próprio nome similar ao nosso português já indica que este é um ambiente onde se pode criar os testes contidos na aplicação que se tornarão a documentação executável da aplicação desenvolvida e por último RoR possui um ambiente - **production** - onde o RoR carrega a classe apenas uma única vez, é onde a aplicação irá rodar em produção constante em um servidor de aplicação, sem a necessidade de sofrer alterações como normalmente é feito em desenvolvimento. Ainda existe uma série de parâmetros que se pode passar para a criação de uma aplicação em rails uma delas é a opção de usar ou não o ActiveRecord que é nativo do rails ao utilizar "**`--skip-active-record`**" o desenvolvedor está explicitamente dizendo ao rails para criar uma aplicação sem persistência à uma base de dados relacional, com isso é possível escolher alguma das bases de dados NoSQL como por exemplo MongoDB, CouchDB, Cassandra etc.

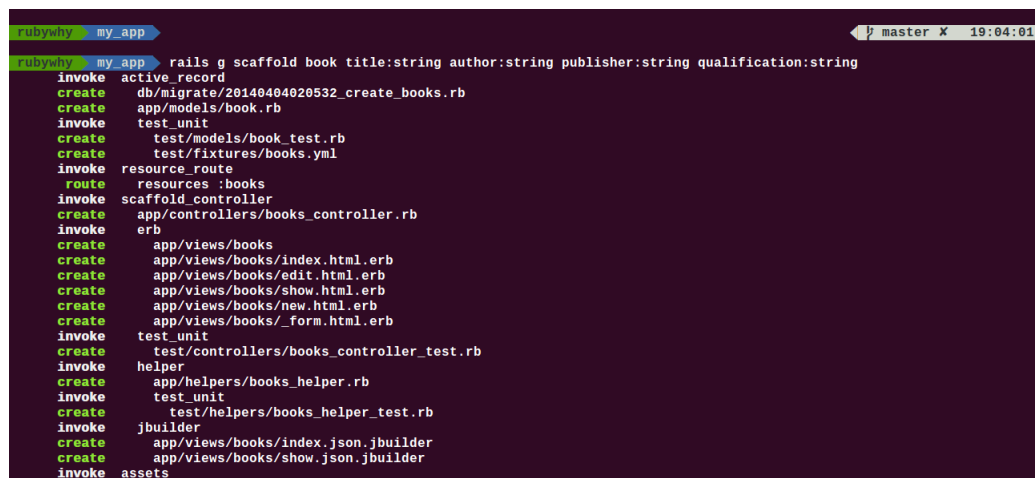
2.3.11 Scaffolding

Rails possui uma ferramenta de desenvolvimento ágil conhecida como scaffold, e essa ferramenta possibilita a criação do Model de acordo com os parâmetros especificados. Este comando possibilita a criação de todo o CRUD, controller, model e views deste modelo dito no ato da criação. Em uma aplicação rails dependendo do que se deseja e dependendo da não complexidade o scaffold se faz uma boa escolha e para utiliza-lo basta usar:

```
>> rails g scaffold user username:string address:string birth_date:date
```

Código 2.9: Exemplo de uso de scaffold

Em uma aplicação rails é sempre bom fazer uso de CoC, e nessas convenções o rails tem uma nomenclatura no que diz respeito a **controllers e models**, toda vez que um controller é criado este sempre estará no plural, já o model correspondente irá sempre estar no singular. Isto quando for utilizado o comando scaffold que no ato irá gerar uma série de outros arquivos além do controller, do model e das views responsáveis por gerenciar o modelo descrito.



```

rubywhy my_app rails g scaffold book title:string author:string publisher:string qualification:string
invoke active_record
create db/migrate/2014040820532_create_books.rb
create app/models/book.rb
invoke test_unit
create test/models/book_test.rb
create test/fixtures/books.yml
invoke resource_route
route resources :books
invoke scaffold_controller
create app/controllers/books_controller.rb
invoke erb
create app/views/books
create app/views/books/index.html.erb
create app/views/books/edit.html.erb
create app/views/books/show.html.erb
create app/views/books/new.html.erb
create app/views/books/_form.html.erb
invoke test_unit
create test/controllers/books_controller_test.rb
invoke helper
create app/helpers/books_helper.rb
invoke test_unit
create test/helpers/books_helper_test.rb
invoke jbuilder
create app/views/books/index.json.jbuilder
create app/views/books/show.json.jbuilder
invoke assets

```

Figura 2.6: Parte dos arquivos criados após o scaffold.

A introdução deste comando *rails g* possibilita ao desenvolvedor uma série de outros comandos uma vez que pode ser abreviado, o nome por extenso para este comando é **generate**. O generate é o gerador de elementos do rails, com ele é possível realizar outras

2.3.12 Comunidade Rails

"A comunidade rubista/rails é hoje uma das mais ativas e unidas do Brasil. Cerca de 10 dos eventos que acontecem anualmente ao redor do mundo com o único propósito de difundir conhecimento e unir os desenvolvedores. Um exemplo dessa força é o Ruby Conf, maior evento de Ruby da America Latina, com presença dos maiores nomes nacionais e internacionais de

Ruby on Rails, e a presença de uma track dedicada ao Ruby na QCon São Paulo."(CAELUM, 2014).

2.4 Git

Git é um sistema de controle de versões distribuídas livre e de código aberto, projetado para lidar com qualquer projeto, desde o menor ao maior com rapidez e eficiência (CHACON, 2009).

A historia do Git está muito relacionada a criação do Linux e de Linus Torvalds, seu criador, bem como com toda comunidade de desenvolvimento Linux. Durante anos a comunidade utilizou a ferramenta *BitKeeper* para guardar a modificações do projeto.

Em 2005, após um problema com a proprietária deste, a comunidade decidiu criar sua própria ferramenta a partir da experiência com a anterior, houve um novo foco em: velocidade, *design* simples, suporte para desenvolvimento paralelo, distribuição completa e a habilidade necessária para lidar com projetos grandes sem perda de velocidade e dados.

Assim, esse novo sistema de versionamento permite que qualquer repositório seja o centro do versionamento, deixando todo *log* das modificações guardados nele sem que para isso precise de uma conexão a rede ou servidor geral.

Para a realização deste projeto foi escolhida uma ferramenta de uso privado conhecida como **bitbucket.org** similar a mais conhecida entre os desenvolvedores o **GitHub**. Esta por outro permite que apenas integrantes do projeto possam opinar bem como realizar alterações de modo mais integro e seguro para aplicações de nível não público.

3 Banco de Dados - SGBDs x NoSQL

Este capítulo procura estabelecer além de uma breve introdução terminológica, um comparativo entre banco de dados relacionais e os NoSQL.

3.1 Breve Histórico

Nos dias de hoje, mediante a demanda elevada de dados trafegados simultaneamente pela rede o mundo, passou a enfrentar uma quantia muito alta e acabando por necessitar de meios mais sofisticados de armazenamento e acesso aos dados. Desde o surgimento do modelo relacional de armazenamento de dados por (CODD, 1970), este tem sido abrangentemente utilizado ao redor do mundo para diversas finalidades. Esse sistema de gerenciamento de banco dados **mundialmente** conhecido como SGBDs, tais como: Mysql, PostgreSQL, Oracle, SQL-Server entre outros foram sendo utilizados ao percurso de muitos anos, sendo estes adotados como padrão por muitas empresas desde comuns até desenvolvedoras de **software**, atendendo completamente ao propósito de cada uma com suas diferentes situações e aplicações, contudo com o avanço das tecnologias empregadas no ramo do desenvolvimento de sistemas web no *século XXI*, com o grande aumento da demanda de dados e com o advento do surgimento das redes sociais, grandes portais com número, intermitente de informações a todo momento, sites de compra coletiva entre outros tipos de sistemas, a arquitetura dos **SGBDs** começou a ser questionada por apresentar certas limitações quanto a escalabilidade e performance. A escalabilidade de um SGBD está relacionada à capacidade de um software crescer de tal forma, rápida e intuitiva e eficiente, que possa atender a uma demanda cada vez maior de dados. A performance do software que faz uso de um SGBDs, por sua vez, faz referência à estimativa do tempo de resposta das requisições efetuadas pelo usuário que faz uso de determinado sistema.

Tendo em vista, o surgimento dessas novas aplicações, fez com que a indústria de desenvolvimento de software apresentasse novas idéias específicas para estas duas situações concistentes e existentes devido às limitações dos sistemas de armazenamento de dados relacionais. As novas aplicações web exploram conteúdo em abrangência na Web 2.0 e diante dos problemas encontrados, foram desenvolvidas novas soluções proprietárias, que diferem do paradigma relacional, as quais foram todas reunidas sob a nomenclatura *NoSQL (Not Only SQL)*.(CUNHA, 2011).

3.2 SGBDs Relacionais

Antes de entrar nas características de uma Base de Dados Relacional, é necessário ter uma noção entre dado, informação e conhecimento. E existe uma ampla diferença entre dados, informação e conhecimento.

- **Dado:** Qualquer fato, instrução formalizada para uso da comunicação, um código para uma obra-prima de objetos, ou seja é a informação não tratada.

"Dados são itens referentes a uma descrição primária de objetos, eventos, atividades e transações que são gravados, classificados e armazenados, mas não chegam a ser organizados de forma a transmitir algum significado específico."(MCLEAN, 2004).

- **Informação:** É a coleção de tudo aquilo que faz sentido e possui valor constitucional e significativo aquelas às quais pode se tirar algum proveito para determinado propósito.

"Informação é todo conjunto de dados organizados de forma a terem sentido e valor para seu destinatário. Este interpreta o significado, tira conclusões e faz deduções a partir deles"(MCLEAN, 2004).

- **Conhecimento:** Segundo (MCLEAN, 2004), assim como conhecimento É a coleção de tudo aquilo que faz sentido e possui valor constitucional e significativo aquelas às quais pode se tirar algum proveito para determinado propósito, conhecimento é a junção de todos os dados e informações processados afim de transmitir a compreensão, experiência, aprendizado acumulativo e alguma técnica, quando aplicada a determinada situação ou atividade que dependa de tal conhecimento.

Em uma base de dados todo e qualquer dado obtido através de um sistema é armazenado e a partir dessa informação contida pode-se obter conhecimento através do processamento dessa coleção de dados e informação. Uma das grandes e principais características do modelo de dados relacionais são as restrições de integridade:

O termo integridade em banco de dados é amplamente utilizado com o seguinte significado: (presisão, correção e validação) e com isso vem o grande problema de muitos sistemas, assegurar que os dados contidos nas tabelas se mantenham precisos, validos e corretos. Visto que se o objetivo do controle de acesso à base de dados é o de evitar que pessoas e/ou programas não autorizados leiam e/ou atualizem ou até mesmo realizem exclusões na base de dados, então é objetivo dos mecanismos que zelam pela integridade semântica do BD garantir que somente atualizações permitidas sejam executadas na base de dados.

Uma outra característica em relação à sistemas criados com base de dados está descrita em condição de sua concistência, disponibilidade, e particionamento como menciona o teorema de CAP.

3.3 Teorema de CAP

Segundo a hipótese da famosa palestra do Drº Erich Brewer, professor da Universidade da Califórnia apresentada no simpósio sobre princípios da computação distribuída (**Principles of Distributed Computing - PODC**) em 2000, ele introduz a afirmativa do teorema e explica que em qualquer sistema distribuído statefull é preciso escolher entre uma forte consistência (C – Consistency), alta disponibilidade (A – availability) e tolerância a particionamento dos dados na rede (P – Network Partition Tolerance) porem nunca os três de uma só vez. Essa afirmação mais tarde em 2002 foi comprovada por **Seth Gilbert e Nancy Lynch** se tornando conhecida como teorema de Brewer ou teorema de CAP. (BREWER, 2000).

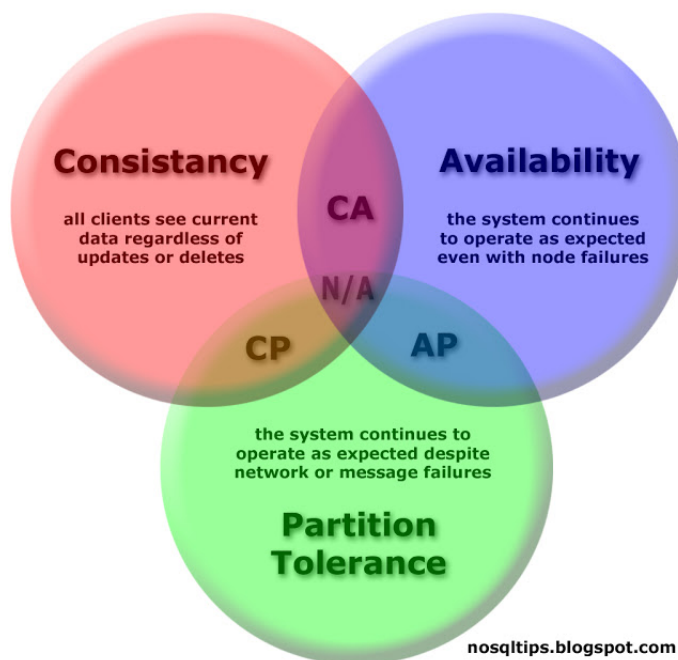


Figura 3.1: Diagrama do Teorema de CAP(Brewer)

3.3.1 C – Consistência

É considerada como a característica que transcreve as duas condicionais de um sistema, como o sistema fica consistente após e uma operação e se ele fica consistente. Uma base de dados consistente distribuída é dita como fortemente consistente ou como tendo fraca consistência. Os sistemas com uma forte consistência, implementam as características ACID (Atomicidade, Consistência, Isolamento e Durabilidade), sendo estas implementadas na maior parte das bases de dados relacionais. Na outra extremidade, de fraca consistência, temos as bases de dados BASE (Basic Availability, Soft-State, Eventual Consistency).

3.3.2 A – Disponibilidade

A disponibilidade que, segundo (BROWNE, 2009), deve garantir que um sistema esteja sempre fornecendo acesso e funcionalidades a seus usuários, é uma das propriedades mais importantes e indispensável para sistemas de empresas, sejam eles baseados em Web ou não mas que precisam disponibilizar seus serviços continuamente, por exemplo, empresas como a Google ou a Microsoft. Entretanto inevitáveis falhas podem ocorrer, e é nesse momento em que alguma contingência (backups automáticos e alguma redundância) deve ocorrer para minimizar o impacto ao usuário.

3.3.3 P – Tolerância ao Particionamento

Tolerância ao particionamento significa assegurar que todas as operações sejam completadas, pode significar ao mesmo tempo a habilidade de um sistema continuar operando, mesmo em situações em que componentes individuais já não estejam mais disponíveis, ou seja onde ocorra uma interrupção parcial de alguns componentes. "Assim, se um sistema disponibilizar essa propriedade, ele ainda será capaz de realizar operações como leitura e escrita mesmo após serem realizados diversos particionamentos na rede".(SETH, 2002; LYNCH, 2002).

3.4 NoSQL

Muitas organizações que tradicionalmente coletam uma quantidade imensa de dados estruturados em bases de dados relacionais, em vista da demanda de dados incessantes crescendo a todo momento, começaram a procurar novas soluções para armazenamento de dados acabando por optarem pelos bando de dados não relacionais, ou bando de dados não-SQL, que agora são frequentemente chamados de banco de dados NoSQL.

O termo NoSQL Database que pode ser entendido como **Not-Only-SQL**, foi inicialmente anunciado em 1998, para bases de dados que omitiam a linguagem SQL, um termo genérico re-introduzido por (EVANS, 2009) da Rackspace, durante um movimento chamado *NoSQL meetup in San Francisco*, promovido Johan Oskarsson da **Last.fm** um site com objetivo de criar uma rádio online agregando uma comunidade virtual com foco em música; o evento foi criado com o intuito de descrever bases de dados que não mais se dispõem do uso de SQL tais como: **Mysql, PostgreSQL, Oracle, SQLServer** (...) em suas transações, sejam elas de manipulação ou consulta de dados. O principal foco era discutir o crescimento e surgimento de novas além de apresentar soluções ou novas alternativas open-source de armazenamento de dados distribuídos não relacionais. Os NoSQL surgiram como uma solução para a questão de escalonamento, consistência e disponibilidade no armazenamento e processamento de grandes volumes de dados na **WEB 2.0**.

"Ao que tudo indica o termo noSQL foi criado em 1998 por Carlo Strozzi para nomear seu projeto open source, que tinha como objetivo ser uma implementação mais leve de um banco de dados relacional, porém sua principal característica era não expor a interface SQL."(PORCELLI, 2011)

A principal característica dos noSQL é o fato de não apresentarem a linguagem SQL, e essa iniciativa de retirar as estruturas impostas pelos SGBDs baseados no modelo relacional de dados, garante que os noSQL em determinadas ocasiões, busquem: acelerar, otimizar e organizar os dados de maneira de objetiva direta e livre de relacionamentos, todavia a diferença mais importante e incessantemente discutida entre as base de dados não relacionais é a da alta escalabilidade necessária para gerenciar enormes volumes de dados, tornando estes mais flexíveis as características **ACID**, onde os atuais banco de dados relacionais são muito restritos quanto a escalabilidade pois fazem uso de escalonamento vertical ou seja quanto mais dados trafegarem mais espaço é solicitado e com isso mais memória é necessário para manter o servidor conciso na maipulação do volume de dados. A diferença única e exclusivamente se encontra no escalonamento, os noSQL fazem uso do escalonamento horizontal e possuem maior facilidade de distribuição de dados, ou seja com o aumento do numero de dados, surge-se a necessidade de aumentar o numero de servidores, otimizando o desempenho e performance do armazenamento.

3.4.1 Escalonamento horizontal

Como fora mencionado na sessão acima sobre escalonamento horizontal, este tende a ser uma solução mais confiável, apesar de possuir alguns requerimentos como diversas threads/-processos distribuidos, ou seja fazendo uma divisão desses dados em varios outros servidores distribuidos, minimizando o volume de dados por servidor, e neste caso menos processamento e gerenciamento de dados e memória são gastos, tornando determinada situação mais simplificada. Neste caso o uso de uma base de dados relacional começa a se tornar inviável, uma vez que com um número de processos distribuidos conectados simultaneamente à um mesmo modelo de dados relacional, causaria uma alta concorrência entre os mesmos, fazendo com que haja um tempo maior de acesso às tabelas envolvidas bem como seus relacionamentos.

"Quando pensamos na arquitetura de sistemas com grande volume de dados a primeira palavra que vem a mente é escalar. Além de desejar que cada uma das pesquisas em nosso sistema execute o mais rápido possível, precisamos criar meios para que, quando necessário, seja fácil adicionar mais recursos (como memória ou novos servidores) e o sistema consiga tirar proveito deles. Para isso muitas vezes precisamos ir além das diversas otimizações de performance e escalabilidade, como por exemplo a criação de um índice para buscas, o uso de caches e de chamadas assíncronas."(ALMEIDA, 2010; SILVEIRA, 2010)

Para que seja permitido o escalonamento horizontal, deve existir a ausência de bloqueio, o que torna essa tecnologia mais adequada e capaz de solucionar problemas quanto ao geren-

ciamento do grande volume de dados que crescem exponencialmente, assim como os dados na WEB 2.0. Existem duas abordagens *replicação e particionamento horizontal* de extrema importância para se conseguir o escalonamento horizontal. Replicação possui uma idéia bastante diversificada por muitos, porem neste caso significa, dividir toda e qualquer parte da base de dados em multiplas partes e as armazenando em mais de um nó na rede. A implementação desta técnica pode muito bem ser criada através de uma clássica estrutura vista em diversos outros sistemas distribuidos a estrutura **master-slave** de processos se comunicando através de troca de mensagens.

Segundo (SLOMAN, 1999), "Um sistema de processamento distribuído é tal que, vários processadores e dispositivos de armazenamento de dados, comportando processos e/ou bases de dados, interagem cooperativamente para alcançar um objetivo comum. Os processos coordenam suas atividades e trocam informações por passagem de mensagens através de uma rede de comunicação".

A aplicação da técnica de escalonamento horizontal, também conhecida como **Sharding**, faz com que as queries possam ser executadas em mais de um nó e quando da sobre-carga dos servidores um novo nó escravo(slave), é criado aliviando os trabalhos ocorridos nos demais nós encontrados na base de dados. Uma vantagem dessa técnica está em caso um desses nó pare ou aconteça algum conflito, não haverá segundo pesquisadores, problemas com as queries, uma vez que outros nós podem responder pelos mesmos processos.

3.4.2 NoSQL 1: Sistemas CP

3.4.3 NoSQL 2: Sistemas AP

3.5 Comparativo entre SGBDS Relacionais e Não Relacionais

4 SGBDS

4.1 Porque Utilizar?

4.2 Banco de Dados Relacionais

4.3 NoSQL

4.4 Comparativo entre SGBDS Relacionais e Não Relacionais

REFERÊNCIAS BIBLIOGRÁFICAS

AKITA, F. *Entendendo Melhor sobre Ruby Gems*. 2009. Disponível em: <<https://www.akitaonrails.com>>. Acesso em: 10/04/2014.

ALMEIDA, A. *quando-muitos-dados-passam-a-atrapalhar-replicacao-e-sharding*. 2010. Disponível em: <<http://blog.caelum.com.br/quando-muitos-dados-passam-a-atrapalhar-replicacao-e-sharding/>>. Acesso em: 03/05/2014.

BAKHARIA, A. *Design Ruby on Rails Power: The Comprehensive Guide*. Boston, MA: Thomson Course Technology, 2007.

BREWER, E. *Teorema de CAP*. 2000. Disponível em: <<http://blog.caelum.com.br/nosql-do-teorema-cap-para-paccl/>>. Acesso em: 19/04/2014.

BROWNE, J. *Brewer's CAP Theorem*. 2009. Disponível em: <<http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>>. Acesso em: 22/04/2014.

CAELUM. *Desenvolvimento Ágil para Web Com Ruby On Rails 4 - RR-71*. www.caelum.com.br/online: Caelum Cursos Online, 2014.

CHACON, S. *Pro Git*. São Paulo: Apress, 2009.

CODD, D. E. F. *Criador dos SGBDs*. 1970. Disponível em: <<http://pt.slideshare.net/Celio12/comparativo-tecnico-entre-tecnologias-de-banco-de-dados-relacional-nosql-newsql>>. Acesso em: 10/04/2014.

CUNHA, T. M. d. A. Escalabilidade de sistemas com banco de dados nosql: um estudo de caso comparativo com mongodb e mysql. 85 f. *Trabalho de Conclusão de Curso (Ciência da Computação)* – Centro Universitário da Bahia – Estácio, Salvador., 2011.

EVANS, E. *NoSQL: What's in a name?* 2009. Disponível em: <http://blog.sym-link.com/2009/10/30/nosql_whats_in_a_name.html>. Acesso em: 02/05/2014.

HANSSON, D. H. *Criador do Rails*. 2006. Disponível em: <http://eu.dbpedia.org/page/David_Heinemeier_Hansson, <https://github.com/dhh>, <http://david.heinemeierhansson.com/about.html>>. Acesso em: 04/03/2014.

LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. 2002.

MATZ. *Ruby-Creator*. 2000. Disponível em: <<https://www.ruby-lang.org/pt/about/>>.

MCLEAN, E. *Tecnologia da Informação para Gestão: Transformando negócios na economia digital*. 3ª ed. Porto Alegre: Bookman Companhia Ed, 2004.

ORSINI, R. *Rails Cook Book*. Boston, MA: O'Reilly Media, 2007.

PORCELLI, A. O que é nosql? *Java Magazine* - ano - VII, n. 86, Janeiro 2011.

RUBY, S. *Agile Web Development With Rails 4*. North Carolina, NC: The Facets, 2014.

SETH, G. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. 2002.

SILVEIRA, G. *quando-muitos-dados-passam-a-atrapalhar-replicacao-e-sharding*. 2010. Disponível em: <<http://blog.caelum.com.br/quando-muitos-dados-passam-a-atrapalhar-replicacao-e-sharding/>>. Acesso em: 03/05/2014.

SLOMAN, M. Network and distributed system management. *IEEE Transactions of software Engineering*, n. 25, November/December 1999.

TECTARGET. *object-relational mapping (ORM)*. 2008. Disponível em: <<http://searchwindevelopment.techtarget.com/definition/object-relational-mapping>>. Acesso em: 29/03/2014.