

Aplicações Web com Ruby On Rails

Com.pensar 2016



PUC Minas
Poços de Caldas

Luiz Alberto Ferreira Gomes

Curso de Ciência da Computação

20 de maio de 2017

Agenda

- 1 Ruby on Rails
- 2 Linguagem Ruby
- 3 Aplicação Exemplo
- 4 The Model
- 5 The Controller
- 6 The View

Ruby on Rails (1)

- Ruby on Rails (Rails) é framework construído na linguagem Ruby para o desenvolvimento de aplicações web
 - Rails é fornecido em uma **gem** Ruby (gem é um pacote Ruby)
- Rails fornece uma extenso conjunto de geradores de código e scripts de automação de testes
- Um conjunto de ferramentas adicionais são fornecidos como parte do ecossistema Rails:
 - **Rake** - utilitário similar ao **make do Unix** para criar e migrar bancos de dados, limpar sessões de uma Web app
 - **WEBrick** - servidor web de desenvolvimento para execução de aplicações Rails

Ruby on Rails (2)

- **SQLite** - um servidor de banco de dados simples pré-instalado como o Rails
- **Rack Middleware** - interface padronizado para interação entre um servidor web e uma Web App
- Algumas empresas que utilizam Rails: Twitter, Hulu, GitHub, Yellow Pages e etc

Agenda

- 1 Ruby on Rails
- 2 Linguagem Ruby**
- 3 Aplicação Exemplo
- 4 The Model
- 5 The Controller
- 6 The View

Histórico da Linguagem Ruby

- Inventada por Yukihiro "Matz" Matsumoto
- Verão 1.0 liberada em 1996(Japão)
- Popularizado no início de 2005 pelo Rails



Visão Geral

- Dinâmica
- Orientada a Objetos
- Elegante, expressiva e declarativa
- Influenciada pelo Perl, Smalltalk, Eiffel e Lisp
- Projeta para tornar o programador "Feliz"

..Java..

```
1 public class Print3Times {  
2     public static void main(String[] args) {  
3         for(int i = 0; i < 3; i++) {  
4             System.out.println("Hello World!")  
5         }  
6     }  
7 }
```

..Ruby..

```
1 3.times {puts "Hello World!"}
```

Básico do Ruby

- Indentação de 2 espaços para cada nível aninhado (recomendado)
- `#` é utilizado para comentários
 - use com moderação, o código deve ser auto documentado
- Scripts utilizam a extensão `.rb`

Listing 1: `codigos/ruby/01-ruby-introducao/introducao.rb`

```
1 3.times { puts "Hello World" }  
2 # um comentario em ruby  
3 puts 5
```

Impressão na Tela

- `puts` é método padrão para impressão em tela
 - insere uma quebra de linha após a impressão
 - similar ao `System.out.println` do Java

Executando um Script Ruby

Convenção de Nomes

■ Variáveis e Métodos

- em minúsculas e separada_por_sublinhado (tenha mais de uma palavra)
- métodos ainda permitem no final os caracteres ?!

■ Constantes

- tanto TODAS_AS_LETRAS_EM_MAIUSCULAS ou no formato CamelCase

■ Classes(e módulos)

- formato CamelCase

Remoção do Ponto-e-Vírgula

- Não coloque o ponto-e-vírgula no final da linha
- Pode ser utilizado para colocar várias declarações em uma linha
 - altamente desencorajado

```
1      a = 3
2      a = 3; b = 5
```

Interactive Ruby (IRB)

- Console interativa para interpretação de comandos Ruby
- Instalado com o interpretador Ruby
- Permite a execução de comandos rapidamente

Filosofia do Rails (1)

- Ruby on Rails é 100% open-source, disponível por meio da MIT License:
⟨<http://opensource.org/licenses/mit-license.php>⟩.
- **Convenção** acima da Configuração (em inglês: *Convention over Configuration* (CoC))
 - se nomeação segue certas convenções, não há necessidade de arquivos de configuração.

Exemplo:

FilmesController#show -> filmes_controler.rb

FilmesController#show -> views/filmes/show.html.e

Filosofia do Rails (2)

- "Don't Repeat Yourself" (DRY) sugere que escrever que o mesmo código várias vezes é uma coisa ruim
- O *Representational State Transfer* (REST) é o melhor padrão para desenvolvimento de aplicações web
 - organiza a sua aplicação em torno de **recursos** e **padrões** HTTP (verbs)

Histórico (1)

- David Heinemeier Hanson **derivou** o Ruby on Rails a partir do BaseCamp – uma ferramenta de gestão de projetos da empresa 37Signals.
 - a primeira versão de código aberto (em inglês: *open source*) foi liberada em julho de 2004.
 - mas direitos para que outros desenvolvedores **colaborassem** com o projeto foram liberados em fevereiro de 2005.
- Em agosto de 2006, o Ruby on Rails atingiu um **marco importante** quando a Apple decidiu distribuído juntamente com a versão do seu sistema operacional Mac OS X v10.5 "Leopard"

Histórico (2)

- nesse mesmo ano o Rails começou a ganhar muita atenção da comunidade de desenvolvimento web.
- Rails é utilizado por diversas companhias, como por exemplo:
 - Airbnb, BaseCamp, Disney, GitHub, Hulu, Kickstarter, Shopify e Twitter.

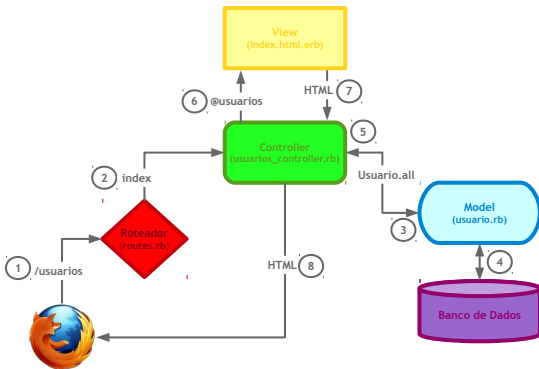
Histórico (3)

Versão	Data
1.0	13 de dezembro de 2005
1.2	19 de janeiro de 2007
2.0	07 de dezembro de 2007
2.1	01 de junho de 2008
2.2	21 de novembro de 2008
2.3	16 de março de 2009
3.0	29 de agosto de 2010
3.1	31 de agosto de 2011
3.2	20 de janeiro de 2012
4.0	25 de junho de 2013
4.1	08 de abril de 2014

Tabela: Evolução histórica do Ruby on Rails

Model-View-Controller

- O framework Rails é contruído em cima do Design Pattern Model View Controller(MVC):



Hora de Colocar a Mão na Massa

- Conecte-se na máquina com o usuário `al550099999` e senha `333333`

1. Inicie uma janela de terminal e digite no prompt:

```
$ rails new my_app
```

2. Mude para o diretório da aplicação (`RAILS.root`)

```
$ cd new my_app
```

3. Execute o servidor web embutido:

```
$ rails s
```

4. Abra uma janela do navegador e digite:

```
$ http://localhost:3000
```

Estrutura de uma Aplicação Rails (1)

Arquivo/Pasta	Descrição
app	Arquivos contendo os principais códigos da aplicação, incluindo modelos, visões, controladores e auxiliares(<i>helpers</i>)
app/assets	Arquivos contendo folhas de estilos (CSS), códigos Javascript e imagens da aplicação
bin	Arquivos ou scripts executáveis
config	Configurações da aplicação
db	Migrações, esquema e outros arquivos relacionados ao banco de dados
doc	Documentação do sistema
lib	Bibliotecas auxiliares
lib/assets	Arquivos contendo folhas de estilos (CSS), códigos Javascript e imagens das bibliotecas

Estrutura de uma Aplicação Rails (2)

Arquivo/Pasta	Descrição
log	Informações de log
public	Páginas que podem ser acessadas publicamente via navegador, tais como páginas de erros
test	Testes da nossa aplicação
tmp	Arquivos temporários como cache e informações de sessões
vendor	Dependências e bibliotecas de terceiros
vendor/assets	Arquivos contendo folhas de estilos (CSS), códigos Javascript e imagens de terceiros
README.rdoc	Uma breve descrição da aplicação
Rakefile	Tarefas que podem ser executadas pelo comando rake
Gemfile	Pacotes(gems) necessários para a aplicação
Gemfile.lock	Uma lista de gems utilizadas para garantir que todas as cópias da aplicação utilizam as mesmas versões de gems
config.ru	Um arquivo de configuração para o Rack Middleware
.gitignore	Define de arquivos ou padrões de arquivos que deverão ser ignorados pelo Git

Agenda

- 1 Ruby on Rails
- 2 Linguagem Ruby
- 3 Aplicação Exemplo**
- 4 The Model
- 5 The Controller
- 6 The View

Especificação do Blog App (1)

1. Blog é uma contração de "weblog", um site de discussão ou troca de informações publicado na Web.
2. Existem dois tipos de participantes: o administrador e o usuário
3. O administrador do blog deve ser capaz de entrar novas postagens, tipicamente em ordem cronológica inversa.
4. Os usuários devem ser capazes de visitar o blog e escrever comentários sobre as postagens.
5. O administrador do blog deve ser capaz de modificar e ou remover qualquer postagem ou comentário.
6. Os usuários não devem ser capazes de modificar postagens ou comentários de outros usuários.

Passos Iniciais do Blog App (1)

1. Inicie uma janela de terminal e digite no prompt:

```
$ cd  
$ rails new blog
```

2. Utilize o gerador scaffold para criar os componentes MVC para as postagens e os comentários

```
$ rails generate scaffold post \  
  title:string body:text  
$ rails generate scaffold comment post_id:int  
  body:text
```

Passos Iniciais do Blog App (2)

3. Gere as tabelas post e comment no banco de dados

```
$ rake db:migrate
```

4. Visualize todas as URLs reconhecidas pela sua aplicação digitando:

```
$ rake routes
```

5. Inicie o servidor web embutido:

```
$ rails s
```

Passos Iniciais do Blog App (3)

6. Abra uma janela do navegador e digite:

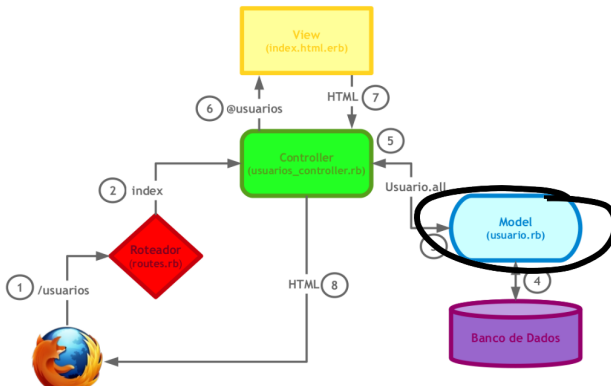
```
$ http://localhost:3000/posts
```

Agenda

- 1 Ruby on Rails
- 2 Linguagem Ruby
- 3 Aplicação Exemplo
- 4 The Model**
- 5 The Controller
- 6 The View

Model Component

- O modelo gerencia os dados, a lógica e as regras de negócios da aplicação.



Banco de Dados Relacionais (1)

- Um aspecto importante da programação web é a habilidade de coletar, armazenar e recuperar diferentes formas de dados
 - uma das formas mais populares são os **bancos de dados relacionais**
- Um banco de dados relacional é baseado entidades, denominadas **tabelas**, no relacionamento, **associações**, entre elas
- O contêiner fundamental em um banco de dados relacional é denominado de **database** ou **schema**
 - podem incluir estruturas de dados, os dados propriamente ditos e permissões de acesso

Banco de Dados Relacionais (2)

- Os dados são armazenados em **tabelas** e as tabelas são divididas em **linhas** e **colunas**. Por exemplo:

Tabela: comment

id	post_id	body
10	1	Ruby realmente...
11	2	Rails facilita...
13	2	Concordo, ...

Banco de Dados Relacionais (3)

- Relacionamentos são estabelecidos entre tabelas para que a consistência dos dados seja mantida em qualquer situação e podem ser:
 - 1:1
 - 1:N
 - N:M

Tabela: comment

id	post_id	body
10	1	Ruby realmente...
11	2	Rails facilita...
13	2	Concordo, ...

Tabela: post

id	title	body
1	A Linguagem Ruby	Ruby é legal.
2	O Framework Rails	O Rails facilita...

Banco de Dados Relacionais (4)

SQLite (1)

- O banco de dados que o Rails utiliza em diversos ambientes (desenvolvimento, teste e produção) é especificado em:
`config/database.yml`

```
1 default: &default
2   adapter: sqlite3
3   pool: 5
4   timeout: 5000
5 development:
6   <<: *default
7   database: db/development.sqlite3
8 test:
```

SQLite (2)

```
9  <<: *default
10 database: db/test.sqlite3
11 production:
12 <<: *default
13 database: db/production.sqlite3
```

- Rails usa por padrão o SQLite como gerenciador padrão
 - relacional, embutido, sem servidor, configuração zero, transacional, suporta SQL

SQLite (3)

ATENÇÃO: SQLite não um banco de dados para produção !

- Banco de dados de produção populares: **MySQL** e **PostgreSQL**

Database Console

- O comando **rails db** fornece uma console para acesso aos bancos dados MySQL, PostgreSQL e SQLite.

```
$ rails db
SQLite version 3.8.7.1 2014-10-29 13:59:56
Enter ".help" for usage hints.
sqlite> .headers on
sqlite> .mode columns
sqlite> select * from posts;
```

id	title	body	created_at	updated_at
5	A Linguagem Ruby	Ruby e legal.	2016-04-30 22:45:20.636363	2016-04-30 22:45:20.636363

```
sqlite>
```

- Dica: utilize **headers on** e **mode coluns**

Hora de Colocar a Mão na Massa (1)

- Inicialize **na pasta da aplicação** a console do banco de dados e configure a sua exibição:

```
$ rails db  
sqlite> .headers on  
sqlite> .mode columns
```

- Exiba os colunas da tabela posts:

```
sqlite> .schema posts
```


Hora de Colocar a Mão na Massa (2)

- Crie um novo post e salve no banco de dados:

```
sqlite> INSERT INTO posts  
(title, body, created_at, updated_at)  
VALUES ("Com.pensar 2016", "Tem varios cursos",  
"2016-05-03 19:50:00", "2016-05-03 19:50:00");
```

- Exiba todos os posts:

```
sqlite> SELECT * FROM posts;
```

- Exiba todos os posts ordenados pelo título (title):

```
sqlite> SELECT * FROM posts ORDER BY title;
```

Hora de Colocar a Mão na Massa (3)

- Exiba um post:

```
sqlite> SELECT * FROM posts LIMIT 1
```

- Exiba o post cujo id é 2:

```
sqlite> SELECT * FROM posts WHERE id=2;
```

- Atualize o título post cujo o id é 2:

```
sqlite> UPDATE posts SET title="Novo titulo"  
WHERE id=2;
```

Hora de Colocar a Mão na Massa (4)

- Remova post cujo o id é 2:

```
sqlite> DELETE FROM posts WHERE id=2;
```

Migrations (1)

- Como podemos rastrear e desfazer alterações em um banco de dados?
- Não existe uma maneira fácil - manualmente é confuso e propenso a erros.
- Tipicamente, comandos SQL são dados para criar e modificar tabelas em um banco de dados
- Mas se houver a necessidade de trocar o banco de dados "durante o voo"?
 - por exemplo, desenvolve-se em SQLite e implanta-se em MySQL.

Migrations (2)

SOLUÇÃO: Migrations

Migrations (3)

- A cada vez que o **scaffold** é executado na aplicação, o Rails cria um arquivo de **migration** de banco de dados. Este arquivo é armazenado em **db/migrate**
- Por exemplo: o arquivo `20160430140114_create_posts.rb`

```
1 class CreatePosts < ActiveRecord::Migration
2   def change
3     create_table :posts do |t|
4       t.string :title
5       t.text :body
6
7       t.timestamps null: false
```

Migrations (4)

```
8   end
9   end
10 end
```

- Rails utiliza o comando **rake** para executar os **migrations** e fazer as alterações no banco de dados.

```
$ rake db:migrate
```

Object-Relational Mapping (1)

- Um ORM **preenche a lacuna** entre banco de dados relacionais e as linguagens de programação orientadas a objetos
- **Simplifica** bastante a escrita de códigos para acessar o banco de dados.
- Tipicamente, comandos SQL são dados para criar e modificar tabelas em um banco de dados
- No Rails, o Model do MVC utiliza algum framework de ORM

Active Record (1)

- ActiveRecord é o nome do ORM padrão do Rails?

Onde está código ?
R: Metaprogramação +
Convenção

Listing 2: app/models/post.rb

```
1 class Post < ActiveRecord::Base
2 end
```

- Para que "mágica" ocorra:
 - o ActiveRecord tem que saber como encontrar o banco de dados (ocorre via config/database.yml)

Active Record (2)

- **(Convenção)** existe uma **tabela** com o **nome no plural** da subclasse `ActiveRecord::Base`
- **(Convenção)** espera-se que a tabela tenha uma chave primário denominada **id**

Object-Relational Mapping (1)

- Um ORM **preenche a lacuna** entre banco de dados relacionais e as linguagens de programação orientadas a objetos
- **Simplifica** bastante a escrita de códigos para acessar o banco de dados.
- Tipicamente, comandos SQL são dados para criar e modificar tabelas em um banco de dados
- No Rails, o Model do MVC utiliza algum framework de ORM

Hora de Colocar a Mão na Massa (1)

- Inicialize **na pasta da aplicação** a console do Rails (não a do banco de dados):

```
$ rails c
```

- Exiba os atributos da classe Post:

```
irb(main):004:0> Post.column_names
```

- Crie um novo post e salve no banco de dados:

Hora de Colocar a Mão na Massa (2)

```
irb(main):005:0> p1 = Post.new
irb(main):006:0> p1.title="Temperatura em Pocos"
irb(main):007:0> p1.body="Esta muito frio..."
irb(main):008:0> p1.save
```

- Exiba todos os posts:

```
irb(main):007:0> Post.all
```

- Exiba todos os posts ordenados pelo título (title):

```
irb(main):007:0> Post.all.order(title: :asc)
```

Hora de Colocar a Mão na Massa (3)

- Exiba um post:

```
irb(main):007:0> Post.first
```

- Exiba o post cujo id é 2:

```
irb(main):007:0> Post.find_by(id: 2)
```

- Atualize o título do primeiro post:

```
irb(main):007:0> p1=Post.first  
irb(main):008:0> p1.update(title: "um novo título")
```

Hora de Colocar a Mão na Massa (4)

- Remova do primeiro post:

```
irb(main):007:0> p1=Post.first  
irb(main):008:0> p1.destroy
```

Validação em Aplicações Web

- **Validação de Dados** é o processo para **garantir** que a aplicação web operem **corretamente**. Exemplo:
 - garantir a validação do e-mail, número do telefone e etc
 - garantir que as "regras de negócios" sejam validadas
- A **vulnerabilidade** mais comum em aplicação web é a **injeção SQL**

Client Side

- Envolve a verificação de que os formulários HTML sejam preenchidos corretamente
 - **JavaScript** tem sido tradicionalmente utilizado.
 - **HTML5** possui "input type" específicos para checagem.
 - Funciona melhor quando combinada com validações do lado do servidor.

Server Side

- A validação é feita após a submissão do formulário HTML
 - **banco de dados**(stored procedure) - dependente do banco de dados
 - **no controlador** - veremos mais tarde que não se pode colocar muita lógica no controlador (controladores magros)
 - **no modelo** - boa maneira de garantir que dados válidos sejam armazenados no banco de dados (database agnostic)
 - Funciona melhor quando combinada com validações do lado do servidor.

Validação em Rails (1)

- **Objetos** em um sistema OO como tendo um **ciclo de vida**
 - eles são criados, atualizados mais tarde e também destruídos.
- Objetos ActiveRecord têm **métodos** que podem ser chamados, a fim de assegurar a sua **integridade** nas várias fases do seu ciclo de vida.
 - garantir que todos os atributos são **válidos** antes de salvá-lo no banco de dados
- **Callbacks** são métodos que são invocados em um ponto do ciclo de vida dos objetos ActiveRecord
 - eles são "ganchos" para gatilhos para acionar uma lógica quando houver alterações de seus objetos

Validação em Rails

- **Validations** são tipo de **callbacks** que podem ser utilizados para garantir a validade do dado em um banco de dados
- Validação são definidos nos **modelos**. Exemplo:

```
1      class Person < ActiveRecord::Base
2          validates_presence_of :name
3          validates_numeracality_of :age, :only_integer
4          validates_confirmation_of :email
5          validates_length_of :password, :in => 8..20
6      end
```

Hora de Colocar a Mão na Massa

- Modifique o arquivo `app/models/post.rb` para exigir que o usuário digite o título e o texto do blog:

```
1      class Post < ActiveRecord::Base
2          validates_presence_of :title, :body
3      end
```

- Modifique o arquivo `app/models/comment.rb` para exigir que o usuário digite texto do comentário blog:

```
1      class Post < ActiveRecord::Base
2          validates_presence_of :body
3      end
```

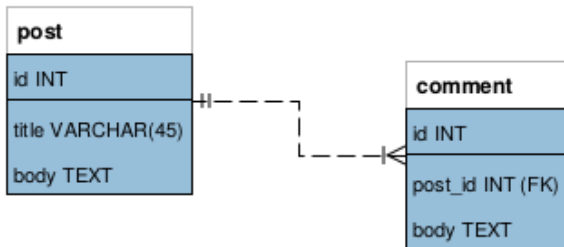
- Inicie o servidor web embutido e teste se a validação está funcionando

Associações em Rails (1)

- O gerador scaffold utiliza por padrão o ActiveRecord. Isto significa:
 - Tabelas para postagens e comentários foram criadas quando executamos as migrações
 - Um conexão com o banco de dados é estabelecida
 - O ORM é configurado para as postagens e comentários foi criado - o "M" do MVC.
- No entanto, uma coisa está faltando:
 - tem-se que assegurar que qualquer comentários sejam associados às suas postagens
- Para tornar os modelos em Rails totalmente funcionais precisamos adicionar **associações**:

Associações em Rails (2)

- cada postagem precisa saber os comentários associado a ele
- cada comentário precisa saber qual é a postagem ele pertence
- Há uma relação **muitos-para-um** entre comentários e postagens uma:



Associações em Rails (3)

- O ActiveRecord contém um conjunto de métodos de classe para **vinculação** de objetos por meio de **chaves estrangeiras**
- Para habilitar isto, deve-se declarar as **associações** dentro dos modelos usando:

Associação	Modelo Pai	Modelo Filho
Um-para-um	has_one	belongs_to
Muitos-para-um	has_many	belongs_to
Muitos-para-muitos	has_and_belongs_to_many	*na tabela junção

Hora de Colocar a Mão na Massa (1)

- Modifique o arquivo `app/models/post.rb` para associar o post aos seus comentário:

```
1      class Post < ActiveRecord::Base
2          validates_presence_of :title, :body
3          has_many :comments
4      end
```

- Modifique o arquivo `app/models/comment.rb` para associar o comentário ao seu post:

Hora de Colocar a Mão na Massa (2)

```
1  class Comment < ActiveRecord::Base
2    validates_presence_of :body
3    belongs_to :post
4  end
```

- Crie um novo post e salve no banco de dados:

```
irb(main):005:0> p1 = Post.new
irb(main):006:0> p1.title="Associacao"
irb(main):007:0> p1.body="Vinculando com o comen
irb(main):008:0> p1.save
```

Hora de Colocar a Mão na Massa (3)

- Crie um novo comment e o vincule a um post:

```
irb(main):005:0> c1 = Comment.new
irb(main):006:0> c1.body="Comentario vinculado"
irb(main):007:0> c1.save
irb(main):008:0> p1.comments << c1
```

- Consulte os comentários do post p1:

```
irb(main):005:0> p1.comments.all
```

- Consulte os comentários 2 do post p1:

```
irb(main):005:0> p1.comments.where(id: 2)
```

Hora de Colocar a Mão na Massa (4)

- Consulte o post do comentário c1:

```
irb(main):005:0> c1.post
```

Agenda

- 1 Ruby on Rails
- 2 Linguagem Ruby
- 3 Aplicação Exemplo
- 4 The Model
- 5 The Controller**
- 6 The View

Action Controller

- Um **Action Controller** é classe Ruby contendo uma ou mais ações
- Cada **ação** é responsável pela resposta a uma requisição
- Quando uma ação é concluída a **visão** de mesmo nome é **renderizada**
- Uma ação deve estar **mapeada** no arquivo **routes.rb** que é gerado pelo scaffold:

```
1 Rails.application.routes.draw do
2   resource :posts
3   resource :comments
4 end
```

Representational State Transfer

- Rails utiliza Representational State Transfer(REST) para mapear os recursos (resources) de uma aplicação:
 - **List** todos os recursos disponíveis
 - **Show** um recurso específico
 - **Destroy** um recurso existente
 - **Provide a way to create** um novo recurso
 - **Create** um novo recurso
 - **Provide a way to update** um recurso existente
 - **Update** um recurso existente

Ação: Index (1)

- Ação que recupera **todas as postagens** do blog
- (Implicitamente) procura pelo template **index.html.erb** para renderizar a resposta

Listing 3: controllers/posts_controller.rb

```
1      Class PostsController < ApplicationController
2
3          # GET /posts
4          # GET /posts.json
5          def index
6              @posts = Post.all
7          end
```


Ação: Index (2)

■ index.html.erb:

Listing 4: views/posts/index.html.erb

```
1      ...
2      <tbody>
3          <% @posts.each do |post| %>
4              <tr>
5                  <td><%= post.title %></td>
6                  <td><%= post.body %></td>
7                  <td><%= link_to 'Show', post %></td>
8                  <td><%= link_to 'Edit', edit_post_path(post) %></td>
9                  <td><%= link_to 'Destroy', post, method: :delete %></td>
```

Ação: Index (3)

```
10         </tr>
11     <% end %>
12 </tbody>
13     . . .
```

Ação: Show (1)

- Recupera **uma** postagem específica no parâmetro **id** passado como parte da URL
- (Implicitamente) procura pelo **show.html.erb** para renderizar a resposta

Ação: Show (2)

Listing 5: controllers/posts_controller.rb

```
1      Class PostsController < ApplicationController
2          before_action :set_post, only: [:show, :edit,
3
4          #GET /posts/1
5          #GET /posts/1.json
6          def show
7          end
8
9          private
10             def set_post
11                 @post = Post.find(params[:id])
```

Ação: Show (3)

12 end

Ação: Show (4)

■ show.html.erb:

Listing 6: views/posts/show.html.erb

```
1      <p>
2          <strong>Title:</strong>
3          <%= @post.title %>
4      </p>
5      <p>
6          <strong>Body:</strong>
7          <%= @post.body %>
8      </p>
9      <%= link_to 'Edit', edit_post_path(@post) %> |
```

Ação: Show (5)

```
10      <%= link_to 'Back', posts_path %> end
```

respond_to (1)

- Rails **helper** que **especifica como responder a uma requisição** baseado no formato da requisição

Listing 7: controllers/posts_controller.rb

```
1 Class PostsController < ApplicationController
2
3   # POST /posts
4   # POST /posts.json
5   def create
6     @post = Post.new(post_params)
7
```


respond_to (2)

```
8     respond_to do |format|
9         if @post.save
10             format.html { redirect_to @post, notice: 'Post was successfully created.' }
11             format.json { render :show, status: :created, location: @post }
12         else
13             format.html { render :new }
14             format.json { render json: @post.errors, status: :unprocessable_entity }
15         end
16     end
17 end
```

redirect_to

- Ao invés de renderizar um template - **envia uma resposta** ao navegador: "go here"
- Usualmente **utiliza uma URL completa** como um parâmetro
 - pode ser tanto uma URL ou uma rota nomeada
- Se o parâmetro é um objeto - Rails tentara **gerar uma URL** para aquele objeto

Ação: Destroy (1)

- Remove uma postagem específica pelo parâmetro **id** passado como parte da URL

Listing 8: posts_controller.rb

```
1 Class PostsController < ApplicationController
2   before_action :set_post, only: [:show, :edit, :update]
3
4   #DELETE /posts/1
5   #DELETE /posts/1.json
6   def destroy
7     @post.destroy
8     respond_to do |format|
```

Ação: Destroy (2)

```
9      format.html {redirect_to posts_url, notice: 'Post destroyed'}
10      format.json {head :no_content}
11  end
12 end
13
14 private
15 # Use callbacks to share common setup or constraints between actions
16 def set_post
17   @post = Post.find(params[:id])
18 end
```

Ação: New (1)

- Cria um novo objeto `post`(vazio)
- (Implicitamente) procura pelo `new.html.erb` para renderizar a resposta

Listing 9: posts_controller.rb

```
1 Class PostsController < ApplicationController
2
3   #GET /posts/new
4   def new
5     @post = Post.new
6   end
```

Ação: New (2)

- `new.html.erb`:

Listing 10: `views/posts/new.html.erb`

```
1 <h1>New Post</h1>
2 <%= render 'form' %>
3 <%= link_to 'Back', posts_path %>
```

Ação: Create (1)

- Cria um novo objeto **post** como os parâmetros que foram passados pelo formulário **new**
- Tenta **salvar** o objeto no **banco de dados**
- Se sucesso, redireciona para o template **show**
- Se insucesso, renderiza o template **new** novamente

Ação: Create (2)

Listing 11: controllers/posts_controller.rb

```
1 Class PostsController < ApplicationController
2   def create
3     @post = Post.new(post_params)
4
5     respond_to do |format|
6       if @post.save
7         format.html { redirect_to @post, notice: 'Post'
8         format.json { render :show, status: :created,
9       else
10         format.html { render :new }
11         format.json { render json: @post.errors, statu
```


Ação: Create (3)

```
12     end
13   end
14 end
15
16 private
17 # Never trust parameters from the scary internet,
18 # only allow the white list through.
19 def post_params
20   params.require(:post).permit(:title, :content)
21 end
```

Ação: Create (4)

- a linha 20 implementa **strong parameters** para aumentar a segurança da aplicação

Hora de Colocar a Mão na Massa

- Modifique o `post_params` para o código abaixo:

Listing 12: `controllers/posts_controller.rb`

```
1 Class PostsController < ApplicationController
2   private
3     # Never trust parameters from the scary internet,
4     # only allow the white list through.
5     def post_params
6       #params.require(:post).permit(:title, :content)
7       params
8     end
```

- tente, agora, criar um post (volte agora para o código original).

Flash (1)

- **Problema:** Queremos **redirecionar** um usuário para uma página diferente do nosso site, mas ao mesmo tempo **fornecer** a ele algum tipo de mensagem. Exemplo: "Postagem criada !"
- **Solução:** flash - uma **hash** onde a dado persiste por exatamente **UMA requisição APÓS** a requisição corrente
- Um conteúdo pode ser colocado em um flash assim:

Listing 13: controllers/posts_controller.rb

```
1      flash[:attribute] = value
```

- Dois atributos **comuns** são **:notice(good)** e **:alert (bad)**

Flash (2)

- Estes dois atributos (:notice ou :alert) podem ser colocados no `redirect_to`
- `show.html.erb`:

Listing 14: views/posts/show.html.erb

```
1      <p id="notice"><%= notice %></p>
2      <p>
3        <strong>Title:</strong>
4        <%= @post.title %>
5      </p>
6      <p>
7        <strong>Body:</strong>
```

Flash (3)

```
8      <%= @post.body %>
9      </p>
10     <%= link_to 'Edit', edit_post_path(@post) %> |
11     <%= link_to 'Back', posts_path %> end
```

Ação: Edit (1)

- Recupera uma postagem específica no parâmetro `id` passado como parte da URL
- (Implicitamente) procura pelo `edit.html.erb` para renderizar a resposta

Ação: Edit (2)

Listing 15: controllers/posts_controller.rb

```
1 Class PostsController < ApplicationController
2   before_action :set_post, only: [:show, :edit, :update]
3
4   #GET /posts/1/edit
5   def edit
6   end
7
8   private
9   def set_post
10    @post = Post.find(params[:id])
11  end
```


Ação: Edit (3)

■ edit.html.erb:

Listing 16: controllers/posts_controller.rb

```
1      <h1>Editing Post</h1>
2      <%= render 'form' %>
3      <%= link_to 'Show', @post %> |
4      <%= link_to 'Back', posts_path %>
```

Ação: Update (1)

- Recupera um objeto **post** utilizando o parâmetro **id**
- Atualiza o objeto **post** com os parâmetros que foram passados pelo formulário **edit**
- Tenta **atualizar** o objeto no **banco de dados**
- Se sucesso, redireciona para o template **show**
- Se insucesso, renderiza o template **edit** novamente

Ação: Update (2)

Listing 17: posts_controller.rb

```
1 Class PostsController < ApplicationController
2   # PATCH/PUT /posts/1
3   # PATCH/PUT /posts/1.json
4   def update
5     respond_to do |format|
6       if @post.update(post_params)
7         format.html { redirect_to @post, notice: 'Post'
8         format.json { render :show, status: :ok, locat
9       else
10        format.html { render :edit }
11        format.json { render json: @post.errors, statu
```

Ação: Update (3)

```
12     end
13   end
14 end
15 private
16   # Never trust parameters from the scary internet,
17   # only allow the white list through.
18   def post_params
19     params.require(:post).permit(:title, :content)
20   end
```

Hora de Colocar a Mão na Massa (1)

- Modifique o arquivo de rotas para aninhar os comentários às postagens e reinicie o servidor:

Listing 18: config/routes.rb

```
1 Rails.application.routes.draw do
2   resources :comments
3   resources :posts do
4     resources :comments
5   end
6 end
```

Hora de Colocar a Mão na Massa (2)

- Crie um novo post e salve no banco de dados:

```
irb(main):005:0> p1 = Post.new
irb(main):006:0> p1.title="Whatsapp bloqueado"
irb(main):007:0> p1.body="A justica bloqueou o W
irb(main):008:0> p1.save
```

- Crie um novo comment e o vincule a um post:

```
irb(main):005:0> c1 = Comment.new
irb(main):006:0> c1.body="O que fazer agora ????"
irb(main):007:0> c1.save
irb(main):008:0> p1.comments << c1
```

Hora de Colocar a Mão na Massa (3)

- Digite no navegador no endereço `<http://localhost:3000/posts/id/comments>`. Onde o **id** é o id do post criado anteriormente.
- Agora crie um novo blog e digite novamente `<http://localhost:3000/posts/id/comments>`. Onde o **id** do blog que acabou de ser criado. (**Temos um problema**)
- Modifique o código do template **views/posts/show.html.erb**. Insira o código abaixo, após a linhas 10 (abaixo do parágrafo do body).

Hora de Colocar a Mão na Massa (4)

```
1 <h2>Comments</h2>
2 <div id="comments">
3   <% @post.comments.each do |comment| %>
4     <%= div_for comment do %>
5       <p>
6         <strong>Posted <%= time_ago_in_words(comment.cre
7         <%= h(comment.body) %>
8       </p>
9     <% end %>
10  <% end %>
11 </div>
```

Hora de Colocar a Mão na Massa (5)

- Agora no navegador visualize uma postagem que tenha comentários.
- Acrescente o código a seguir logo abaixo do código anterior no arquivo `views/posts/show.html.erb`:

```
1 <%= form_for([@post, Comment.new]) do |f| %>
2 <p>
3 <%= f.label :body, "New Comment" %><br>
4 <%= f.text_area :body %>
5 </p>
6 <p>
7 <%= f.submit "Add Comments" %>
8 </p>
```

Hora de Colocar a Mão na Massa (6)

```
9 <% end %>
```

- Modifique a ação create do controlador
`controllers/comments_controller.rb`:
-

```
1 def create
2   @post = Post.find(params[:post_id])
3   @comment = @post.comments.create(comment_params)
4
5   respond_to do |format|
6     if @comment.save
7       format.html { redirect_to @post, notice: 'Commen
8       format.json { render :show, status: :created, lo
```

Hora de Colocar a Mão na Massa (7)

```
9     else
10         format.html { render :new }
11         format.json { render json: @comment.errors, status: :unprocessable_entity }
12     end
13 end
14 end
```

- Escolha uma postagem qualquer e escreva alguns comentários.
- Remova a rota absoluta para comentários no arquivo de rotas e reinicie o servidor:

Hora de Colocar a Mão na Massa (8)

Listing 19: config/routes.rb

```
1 Rails.application.routes.draw do
2   #resources :comments
3   resources :posts do
4     resources :comments
5   end
6 end
```

Agenda

- 1 Ruby on Rails
- 2 Linguagem Ruby
- 3 Aplicação Exemplo
- 4 The Model
- 5 The Controller
- 6 The View**

Action View

- Arquivo HTML com a extensão **.erb**
 - ERb é uma **biblioteca** que permite a colocação de código Ruby no HTML
- Dois padrões a aprender:
 - `<% ...código ruby..%>` avalia o código Ruby
 - `<%= ...código ruby..%>` retorna o resultado do código avaliado

Partials (1)

- Rails encoraja o princípio **DRY**
- O layout da aplicação é mantida em um único local no arquivo **application.html.erb**
- O código comum dos templates ser reutilizado em **múltiplos templates**
- Por exemplo, os formulários do **edit** e do **new** - são realmente muito diferentes ?
- Partials são similares aos templates regulares, mas eles possuem capacidades mais **refinadas**
- Nomes de partials começam com **underscore** (**_**)

Partials (2)

- Partials são renderizados com `render 'partialname'` (sem underscore)
- `render` também aceita um segundo argumento, um hash com as variáveis locais utilizadas no partial
- Similar a passagem de variáveis locais, o `render` pode receber um objeto
- `<%= render @post %>` renderizara `app/views/posts/_posts.html.erb` com o conteúdo da variável `@post`
- `<%= render @posts %>` renderiza uma coleção e é equivalente a:

Partials (3)

Listing 20: controllers/posts_controller.rb

```
1      <% @posts.each do |post| %>
2          <%= render post %>
3      <% end %>
```

Partials (4)

■ `_form.html.erb`

Listing 21: `views/posts/_form.html.erb`

```
1 <%= form_for(@post) do |f| %>
2   <% if @post.errors.any? %>
3     <div id="error_explanation">
4       <h2><%= pluralize(@post.errors.count, "error") %>
5         prohibited this post from being saved:</h2>
6       <ul>
7         <% @post.errors.full_messages.each do |message|
8           <li><%= message %></li>
9         <% end %>
```

Partials (5)

```
10     </ul>
11   </div>
12   <% end %>
13
14   <div class="field">
15     <%= f.label :title %><br>
16     <%= f.text_field :title %>
17   </div>
18   <div class="field">
19     <%= f.label :body %><br>
20     <%= f.text_area :body %>
21   </div>
```

Partials (6)

```
22 <div class="actions">
23   <%= f.submit %>
24 </div>
25 <% end %>
```

Form Helpers (1)

- `form_for` gere a tag form para o objeto passado como parâmetro
- Rails utiliza a método `POST` por padrão
- Isto faz sentido:
 - uma password não é passada como parâmetro na URL
 - qualquer modificação deverá ser feita via POST e não GET

Listing 22: views/posts/_form.html.erb

```
1  <%= form_for(@post) do |f| %>
2    ...
3  <% end %>
```

f.label

- Gera a tag HTML **label**
- A descrição pode ser **personalizada** passando um segundo parâmetro

```
1 <div class="field">
2   <%= f.label :title, "Titulo" %><br>
3   <%= f.text_field :title %>
4 </div>
```

f.text_field

- Gera o campo `input type="text"`
- Utilize `:placeholder` para mostrar um valor dentro do campo

```
1 <div class="field">
2   <%= f.label :title, "Titulo" %><br>
3   <%= f.text_field :title, placeholder: "Escreva o tit
4 </div>
```

f.text_area

- Similar ao `f.text_field`, mas gera um text area de tamanho (40 cols x 20 rows)
- O tamanho pode ser modificado através do atributo `size`:

```
1 <div class="field">
2   <%= f.label :body, "Conteudo" %><br>
3   <%= f.text_area :body, size: "10x3" %>
4 </div>
```

Outros Form Helpers

- `date_select`
- `search_field`
- `telephone_field`
- `url_field`
- `email_field`
- `number_field`
- `range_field`

f.submit

- Renderiza o botão **submit**
- Aceita o **nome** do botão submit como primeiro argumento
- Se o nome não for fornecido - gera um baseado no modelo e na ação. Por exemplo: "Create Post" ou "Update Post"

```
1 <div class="actions">
2   <%= f.submit "Postar"%>
3 </div>
```

- Mais form helpers:
(http://guides.rubyonrails.org/form_helpers.html)