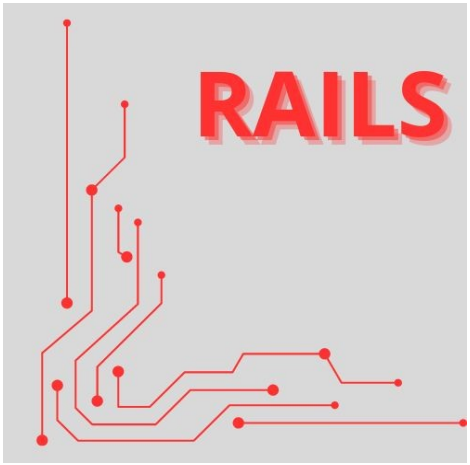
Introdução ao desenvolvimento de API's com

RAILS 7

este e-book possui app Android

Adyson Lima

Sumário



Capítulo 1

Qual o seu propósito ao ler este tutorial?

Como prática comum nos tutorias que escrevo, sempre tiro algumas linhas para perguntar ao leitor sobre o seu propósito. E neste tutorial, se o leitor me permitir, farei o mesmo, com o objetivo de motivar uma breve reflexão. Sendo assim ...

Qual o seu propósito ao ler este tutorial? Aprender uma nova habilidade para usar no seu trabalho? Passar o tempo lendo algo diferente? Ou quem sabe seu propósito seja algo mais profundo como por exemplo:

“ Meu propósito ao ler este tutorial é aprender uma nova habilidade e usar essa habilidade para resolver problemas reais no mercado de trabalho, ser respeitado e bem pago por isso. Dedicar 10% do meu ganho financeiro para divulgar as tecnologias que uso e acredito e além disso, ajudar pessoas em situação de morador de rua, fechando assim, meu ciclo de ação e realização profissional :P ”

Profundo hein? Brincadeiras a parte, o fato é que, **você deve ter um propósito claramente definido**, para impulsionar você para a **ação**. Pois a falta de um propósito, pode levar você a ter resultados abaixo do que você espera. Dito isso, sugiro que você pense alguns minutos sobre o seu propósito e responda a pergunta seguinte.

Qual o meu propósito ao ler este tutorial?

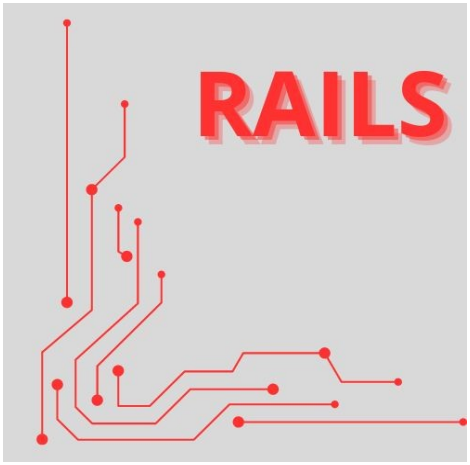
_____

_____

_____

Caso sinta necessidade de um auxílio, este app de minha autoria talvez ajude.

Em seguida, avance para o próximo capítulo.



Capítulo 2

Uma palavra sobre vendas

Por que falar sobre vendas em um tutorial de programação? Bem, embora seja realmente incomum essa prática, sinto que é necessário lembrar o leitor que vendas ocorrem o tempo todo, por exemplo:

- *Vende-se ideias na internet.*

- *Vende-se pães na panificadora.*

- *Vende-se características pessoais e profissionais no Linkedin.*

- *Vende-se serviços.*

- *Vende-se produtos.*

E assim, é preciso lembrar que **softwares são vendidos** também, como API's por exemplo.

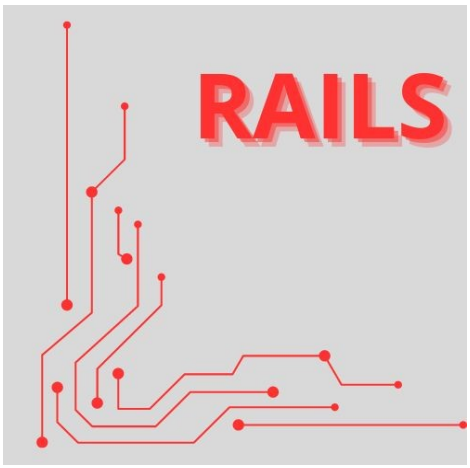
O leitor pode ter em mente que existem diversas maneiras de realizar vendas, de vendas no Instagram, até a clássica, mas ainda viva, venda através de panfletos.

Pense a respeito, você já pediu uma pizza ou outro prato de uma empresa que você tomou conhecimento através do Instagram ou de um panfleto?

Então, sugiro que o leitor busque aprender um pouco sobre vendas, existem excelentes materiais disponíveis, dê uma pesquisada no google, amazon, e você vai encontrar.

O investimento trará retorno com certeza. Pois **a habilidade de vender transcende áreas de conhecimento**, e uma vez aprendida, você poderá usá-la em diversas situações, de uma entrevista de emprego a uma transação contratual com um cliente. Além de ser uma profissão a mais na sua caixa de ferramentas.

E não fique surpreso de um dia ser contratado(a) como desenvolvedor, por alguém que leu um panfleto seu ou clicou em um anúncio seu no Instagram.



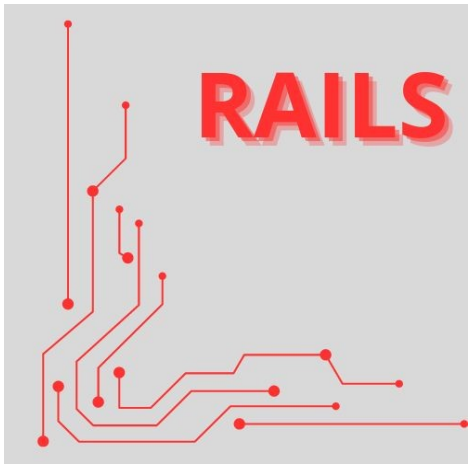
Capítulo 3

Para quem é destinado este material?

Este tutorial, é destinado principalmente a desenvolvedores Ruby, que queiram conhecer um meio de desenvolver Web API’s em Ruby on Rails, usando TDD. No entanto, pode ser lido por qualquer um que tenha interesse.

E mesmo desenvolvedores de software que usam outras tecnologias, podem ler este tutorial e assim, incrementar seus conhecimentos, e ainda ver como desenvolver em Ruby usando Rails pode ser gratificante.

Atenção: a partir de certo momento, a fonte dos códigos neste tutorial mudara de preto para um estilo colorido, isto é **intencional**, para quebrar a monotonia caso ocorra.



Capítulo 4

Requisitos

Existem alguns requisitos necessários para tirar o melhor proveito deste tutorial, e creio que você seja desenvolvedor Ruby caro leitor e assim, você provavelmente já os tem. No entanto, segue a lista do que é necessário.

- Ruby 3 instalado.

- Rails 7 instalado.

- RSPEC instalado.

- VSCode instalado.

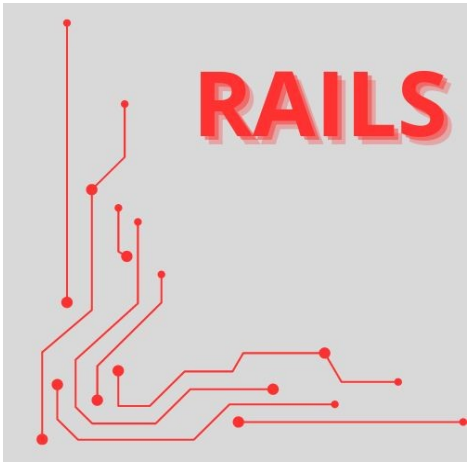
- Git instalado.

- Conta no GitHub.

- Conhecimento em lógica de programação.

- Conhecimento básico de desenvolvimento web.

Caso você não tenha os requisitos citados, você pode muito facilmente, encontrar de maneira abundante na internet fontes que podem suprir sua necessidade. Como por exemplo, o canal do Youtube, puts dev, do Xita. Só para dar um exemplo, existe nesse canal, uma série de vídeos ensinando a desenvolver um framework WEB do zero em Ruby, então, se você deseja evoluir em Ruby . . .



Capítulo 5

Introdução

Desenvolver software é uma atividade que **demanda tempo**, e que **consome recursos mentais** do desenvolvedor. Nesse sentido, é muito importante ter ferramentas que reduzam o tempo de desenvolvimento e que consumam menos recursos mentais do programador. Por questões obvias, como, redução de **custo financeiro** para a empresa e menor desgaste da **saúde** do desenvolvedor.

Assim, surge este tutorial sobre Ruby on Rails, que é um framework Web super maduro, rico em recursos, seguro, estável e consagrado no mundo como uma das melhores opções para desenvolvimento Web.

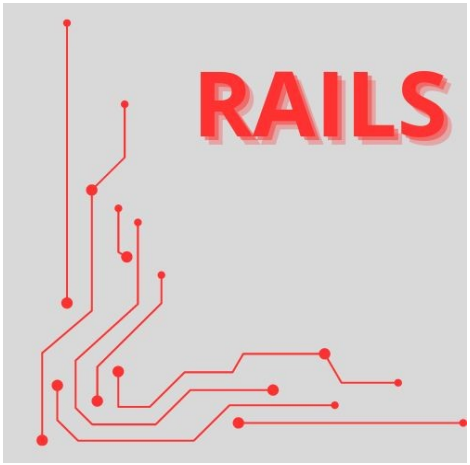
Este tutorial vai mostrar que é menos desgastante o processo de desenvolvimento usando Rails. Tanto pelas convenções usadas por padrão no Rails, que agilizam e reduzem muito o que é necessário configurar no projeto, tanto como pela quantidade de recursos disponíveis facilmente instaláveis.

E além disso, este tutorial irá mostrar como desenvolver usando testes no Rails. E sendo mais especifico, este tutorial vai abordar o TDD, já que esse é um padrão no mercado.

Uma revisão de Ruby será mostrada, com o foco no desenvolvimento com Rails. Assim como uma breve introdução a REST .

O desenvolvimento em Rails será no estilo mão na massa.

Então, sem mais delongas, vamos ao próximo capítulo.



Capítulo 6

Ruby

Ruby é uma linguagem de programação orientada a objetos, criada para ser fácil de usar, divertida e principalmente, poderosa. Objetivos que ela atende muito bem. Criada em 1995, por Yukihiro "Matz" Matsumoto, no Japão, é a base de desenvolvimento usada neste tutorial.

Assim, vamos para uma breve revisão sobre Ruby.

- Variáveis

A ideia de variáveis em Ruby, é semelhante a encontrada em muitas outras linguagens de programação, ou seja, variáveis são um espaço em memória onde é possível "guardar" uma informação de modo temporário. A sintaxe básica de uma variável é mostrada abaixo.

nome_da_variável = valor

Observe que em Ruby, não é utilizado ponto e virgula no final de instruções. E lembre que em Ruby, o tipo da variável é determinado pelo valor que é armazenado nela. Se o valor for inteiro, então a variável é do tipo inteiro, se o valor guardado for uma string, a variável é do tipo string e assim por diante. Vejamos mais alguns exemplos de uso de variáveis em Ruby.

idade = 20 # variável idade recebe um número

meu_nome = "josé" # variável meu_nome recebe um texto

motor_ligado = false # variável motor_ligado recebe um valor booleano

Em Ruby, existem também, variáveis de instância, que são variáveis que estão disponíveis para cada instância de um classe Ruby. Perceba o uso do arroba antes do nome desse tipo de variável. Veja um exemplo dessas variáveis usado no Rails.

@products = Products.all # variável de instância @products recebe todos os produtos

- Operações matemáticas

Realizar operações matemáticas é muito simples em Ruby, tal como os exemplos seguintes.

numero_um = 1 # variável numero_um recebe 1

numero_dois = 2 # variável numero_dois recebe 2

soma = numero_um + numero_dois # variável soma recebe a soma dos números

resto = numero_dois – numero_um # variável resto recebe a diferença dos números

produto = numero_um * numero_dois # variável produto recebe o produto dos números

quociente = numero_dois / numero_um # variável quociente recebe a divisão dos números

- Estruturas de decisão

Em Ruby, são usadas as estruturas **if**, **if else**, **elsif**, **unless**, **case**, **when**, e com exceção das três últimas, seu funcionamento é como em outras linguagens. Vejamos alguns exemplos simples de **if else**, em Ruby puro e dentro de uma aplicação Rails.

```
idade = 18 # variável idade recebe o número 18
if idade <= 17 # se a idade for menor ou igual a 17
  puts 'de menor' # mensagem se a condição if for verdadeira
elsif idade > 17 and idade < 65 # se a idade estiver entre 18 e 64 anos
  puts 'de maior' # mensagem se a condição elsif for verdadeira
else # caso as avaliações if e elsif não sejam verdadeiras
  puts 'senhor(a)' # mensagem para idade maior que 64 anos
end # fim do bloco
```

```
if @bread.update(bread_params) # se a função update executar coretamente
  render json: @bread #  renderize @bread como json
else # do contrário
  render json: @bread.errors, status: :unprocessable_entity # renderize os erros e o status
end # fim do bloco
```

- Loops

Executar loops em Ruby, especialmente dentro de aplicações Rails, que é nosso foco principal neste tutorial, é uma tarefa simples, mas um pouco diferente do usual em outras linguagens. Como podemos ver no exemplo abaixo em um arquivo **seeds.rb** do Rails, o uso de um loop que executa 5 vezes a instrução de criação de um registro no banco de dados. Note que não foi usado um laço **for**, como em outras linguagens de programação.

```
5.times do # executa 5 vezes a criação de um objeto
  Bread.create(name: ['frances', 'hamburger', 'doce'].sample, price: 'preço tabelado')
end
```

- Classes

A declaração de classes, uma das bases da orientação a objetos, é feita em Ruby de acordo a sintaxe abaixo, correspondente a um controller e um model de uma aplicação Rails respectivamente.

```
class BreadsController < ApplicationController


end


class Bread < ApplicationRecord


end
```

Acima, foi declarado o model **Bread**, que é uma classe. E essa classe, herda de **ApplicationRecord**. O sinal de " < " indica a relação de herança.

- Objetos

Objetos em Ruby, como em muitas outras linguagens, são o recurso usável de uma classe, ou uma instância, se preferir. A sintaxe básica para criar um objeto em Ruby é mostrada abaixo. Observe o uso da função **new**.

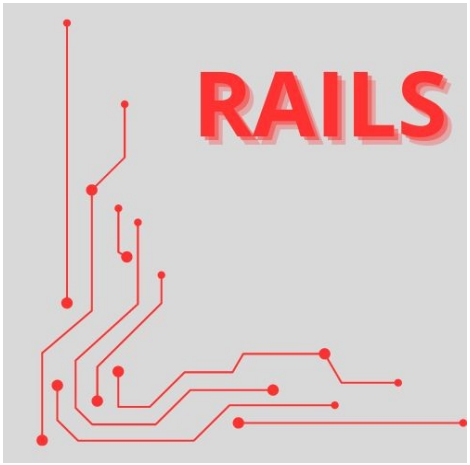
```
bread = Bread.new # objeto bread criado com a função new
```

Em Ruby, os objetos criados tem acesso aos recursos disponibilizados pela classe da qual são instanciados, como em outras linguagens.

- Juntando tudo em um pequeno programa Ruby

Agora, para finalizar essa breve seção de revisão sobre Ruby, vamos juntar as peças; Variáveis, estrutura if else, classes e objetos, em um pequeno programa. Os comentários dão uma explicação sobre o código de forma breve.

```
class Framework

 @name = ' ' # variável que armazena um nome de framework

 # função que salva um nome de um framework
 def save_name
  puts 'digite o nome de um Framework!'
  @name = gets.chomp
 end

 # função que imprime o nome de um framework
 def print_name
  if @name == 'Ruby on Rails'
   puts " #{@name} é baseado em Ruby! "
  else
   puts " #{@name} pode não ser baseado em Ruby! "
  end
 end

end

framework = Framework.new # instanciação da classe Framework

# chamada de duas funções da classe Framework
framework.save_name
framework.print_name
```



Capítulo 7

Ruby on Rails

Finalmente vamos falar sobre o framework para desenvolvimento web mais poderoso, e mais fácil de usar, Ruby on Rails, desculpe a empolgação :)

E para começar, vamos observar uma ideia presente no site “ https://rubyonrails.org/doctrine ”, escrita pelo DHH que diz muito sobre a ferramenta Ruby on Rails.

(*Provide sharp knives, ruby includes a lot of sharp knives in its drawer of features. Not by accident, but by design ...*)

Traduzindo,

(*Forneça* ***facas afiadas****, ruby inclui muitas facas afiadas em sua gaveta de recursos.* ***Não por acidente****, mas por design ...*)

Ruby, e Ruby on Rails trazem um conjunto de recursos extremamente úteis que agilizam e facilitam muito o trabalho de desenvolvimento. Como por exemplo, que tal um banco de dados pronto para uso, com distinção entre ambientes de teste, desenvolvimento e produção, e com uma ferramenta de ORM e um script para popular o banco rapidamente, incluídos por padrão com apenas um comando durante a criação de um projeto?

Bem, Ruby on Rails provê essa facilidade e creio que isso confirma a afirmação sobre “facas afiadas”.

E o que dizer sobre o excesso de poder presente no comando **scaffold**? Para os que não conhecem, esse único comando cria um **crud** completo, tanto para aplicações fullstack quanto para api’s web.

E nem preciso dizer que caso você precise entregar algo muito rapidamente, você pode usar esse comando, **scaffold**. E associar a ele o uso do Bootstrap e fazer o deploy no Heroku. Claro que isso é para casos extremos, pois você não poderá fazer o ciclo TDD usando o **scaffold**, embora o **scaffold** gere testes e manualmente você possa criar seus testes depois também se quiser, mas fica aí a ‘faca afiada’. “ Grandes poderes trazem grandes responsabilidades Peter ”, desculpe ;)

Agora, vamos partir para um pouco de prática com Ruby on Rails.

Tendo em mente que você já tem o Ruby e o Ruby on Rails instalados, abra um terminal no seu sistema, e digite o comando a seguir para criar um projeto.

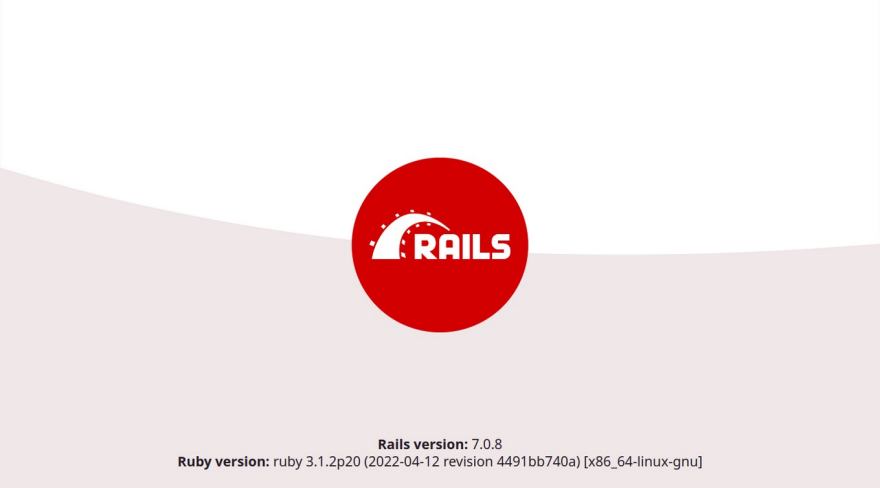
$ rails new meu_projeto --api

Em seguida, entre na pasta do projeto e rode a aplicação com os comandos a seguir.

$ cd meu_projeto

$ rails s

Abra o navegador web e acesse **localhost:3000**, e você verá a tela de boa vindas do Rails, como na imagem a seguir.



Perceba o fato de que você precisou apenas de um comando para criar o projeto, e não precisou de várias configurações. Agora, vamos ver o que o comando **scaffold** oferece. Então, pare a aplicação pressionando a combinação de teclas **ctrl + c** no seu teclado, e digite os comandos abaixo.

$ rails g scaffold Car name age

$ rails db:migrate

Os dois comandos acima geraram pra você, uma api web, pronta para uso, que já pode ser testada usando o Postman, e que já atende requisições feitas por apps Android ou React por exemplo, bastando configurar o **cors** e acessar os endpoints criados, que são rotas disponíveis, lembra das ‘facas afiadas’ ?

O primeiro comando, **rails g scaffold Car name age**, gerou um Crud completo, com a estrutura da tabela **cars**, pronta para ser instalada no banco de dados, e incluiu nessa estrutura, os campos **name** e **age** para manipulação de dados.

O segundo comando, **rails db:migrate**, configurou o banco de dados e o deixou pronto para uso. Agora, vamos configurar o arquivo **seeds.rb**, para gerar alguns registros no banco de dados e fazer alguns testes. Abra o arquivo **meu_projeto/db/seeds.rb** com seu editor de texto favorito e acrescente o código a seguir.

puts ‘gerando carros … ’

5.times do

Car.create(name: [‘opala’, ‘maverick’, ‘dart’**].sample, age:** ‘mais de 30 anos’**)**

end

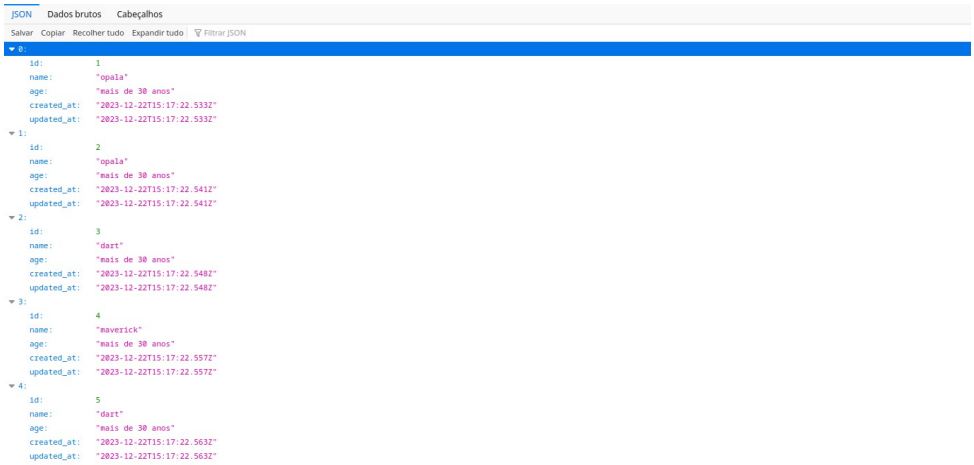
puts ‘carros gerados com sucesso!’

Agora, rode os comandos a seguir para executar as instruções do arquivo **seeds.rb**, e incluir 5 registros no banco de dados. E além disso, iniciar a aplicação novamente.

$ rails db:seed

$ rails s

Agora, acesse a aplicação no navegador web, ou no Postman, com o endereço **http://localhost:3000/cars**, e você verá a lista de carros gerada, como imagem a seguir.



Agora, observe que além da facilidade em criar a API anterior, você tem muitos conceitos importantes e muito úteis aplicados automaticamente durante a criação do projeto. Veja a estrutura do projeto Rails criado em seu editor de texto e atente para:

app

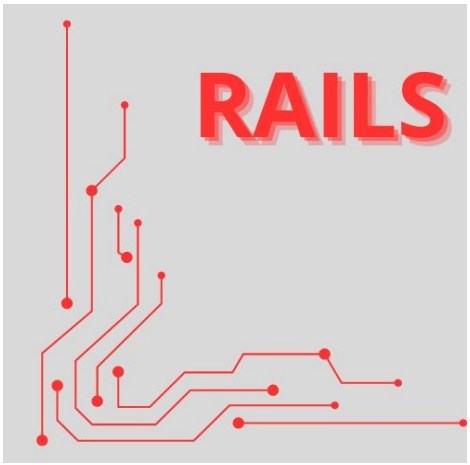
controllers

models

view

Padrão MVC, aplicado

Concluímos este capítulo agora, com uma pequena web api, desenvolvida com Rails. E creio que até aqui você já pode ver o poder, a facilidade e a agilidade que o Rails oferece. Este foi um rápido primeiro contato, vamos nos aprofundar mais adiante.



Capítulo 8

TDD

TDD ou Test Driven Development, é uma técnica de programação que basicamente força o desenvolvedor a escrever código de melhor qualidade e que se utiliza de testes para assegurar essa qualidade. Usando TDD, o programador primeiramente escreve um teste, antes de desenvolver uma funcionalidade.

Mas não se trata apenas de ter mais qualidade e uma aplicação testada, TDD dá ao programador algo fundamental; "**paz na hora de refatorar**". Imagine que você é solicitado a fazer uma alteração em um código e antes de qualquer coisa, você roda os testes da aplicação que foi feita com TDD e vê que todos os testes passam. Isso significa que o que quer que possa dar errado enquanto você refatora, é apenas do último commit de teste até o momento atual. Não há necessidade de entrar em pânico e olhar dezenas, talvez centenas de linhas anteriores. E essa "**paz na hora de refatorar**" se reflete em menos stress e mais saúde. Além de representar entregas mais rápidas.

Com TDD, o desenvolvedor ganha, a empresa ganha e o cliente ganha.

Dito isso, vejamos um pequeno exemplo de aplicação de teste. Imagine que um cliente deseja que um botão some dois números, e essa funcionalidade será entregue com uso de TDD, nesse caso. O desenvolvedor primeiro escreve um teste, como a seguir:

```
describe ‘testes sobre botão de soma de números’ do # descrição do grupo de testes
  it ‘Consegue somar dois números?’ do # descrição de um teste especifico
    # é esperado que a função de somar retorne o resultado dentro de eq( )
    expect(Classe.funcao_soma(numero1, numero_2)).to eq(resultado_da_soma)
  end
end
```

E em seguida, o teste é rodado, retornando um erro, já que não há ainda a função "**funcao_soma(numero1, numero_2)"** para realizar a soma e devolver o resultado. Então, só agora, com um teste escrito e falhando é que será desenvolvida a função de soma.

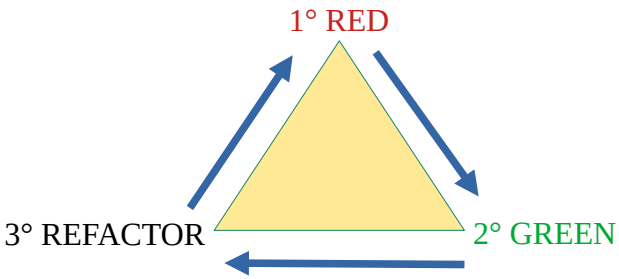
```
def funcao_soma(numero1, numero2)
  numero1 + numero2
end
```

Feita a função, o teste é rodado novamente, desta vez, passando com sucesso. Perceba que foi o teste quem guiou o desenvolvimento da funcionalidade, ele veio primeiro e de certo modo estabeleceu como deveria ser o comportamento da função de soma. E esse resumidamente é o ciclo

de desenvolvimento usando TDD, existe ainda a etapa de refatoração, que consiste em melhorar o teste, mas veremos em mais detalhes isso adiante.

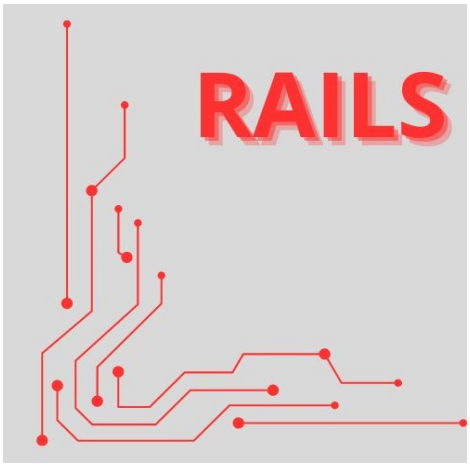
- Etapas do TDD

O TDD possui 3 etapas, **RED**, **GREEN**, **REFACTOR**. A primeira, red, consiste em se escrever um teste simples que vai falhar. A segunda, green, consiste em escrever uma funcionalidade que faça o teste passar. E a terceira, refactor, consiste em melhorar seu teste. A imagem a seguir mostra a ideia de TDD.



O Ruby on Rails traz o minitest como ferramenta padrão de testes, mas também pode ser usado o RSPEC, com uma simples configuração que veremos mais a frente. Você pode acessar o rails guides e ver um pouco sobre o minitest. Mas neste tutorial, vamos usar o RSPEC.

Assim, finalizamos essa breve revisão sobre TDD e testes, vamos agora, seguir para o próximo capítulo.



Capítulo 9

REST

Representational State Transfer, ou REST, é um conceito intimamente ligado ao desenvolvimento de WEB API’s. Este conceito, define como usar métodos HTTP para realizar comunicações entre diferentes aplicações na internet.

Basicamente existem 4 métodos HTTP, que são usados para a comunicação, são eles:

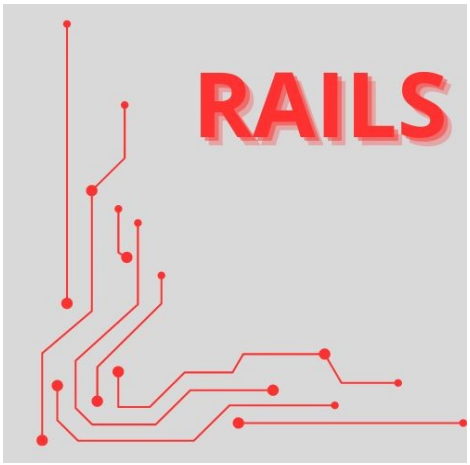
GET : Lista registros.

POST : Insere registros.

PATCH : Atualiza registros.

DELETE : Apaga registros.

Neste tutorial, usaremos os quatro, para realizarmos nossas requisições.



Capítulo 10

Criando uma API com TDD

Finalmente chegamos ao foco principal deste material, o desenvolvimento de um API com Ruby on Rails 7. Sugiro fortemente que você digite os códigos que se seguirão, não se contente com a simples leitura. Lembra do capítulo sobre propósito?

Neste tutorial, vamos criar uma pequena API, usando TDD e Rails 7, e o nosso projeto será um CRUD sobre consoles de videogame, seu nome será, “consoles_api”. O foco desta aplicação é o entendimento de como trabalhar com os conceitos anteriores na prática . Portanto, vamos manter as coisas acessíveis, pois um dos públicos-alvo deste tutorial é programadores que estão buscando se familiarizar com o TDD, API’s e Rails.

Dito isso, vamos começar.

- Criando o projeto

A primeira ação é criar o projeto, o que pode ser feito com o comando a seguir:

$ rails new consoles_api --api --skip-test

O comando anterior cria um projeto Rails, com o nome “consoles_api”. E as flag “--api” diz que o projeto será uma API, e sendo assim, o Rails configura o projeto como tal, por exemplo, não instalando recursos voltados para frontend, o que deixa também o projeto mais enxuto.

A flag “--skip-test”, diz ao Rails para não instalar o framework de testes padrão, isso é necessário pois vamos usar o "rspec" como framework de testes. Pronto, nosso projeto está criado, vamos enviar ele para o GitHub e em seguida, configurá-lo.

- Enviando o projeto para o GitHub

Com o projeto criado, você já pode entrar no seu Github, e enviar seu projeto. Quando criar seu projeto no Github, será mostrada as instruções abaixo que dizem como proceder para enviar o seu projeto:

```
echo "# seu_projeto_api" >> README.md

git init

git add README.md

git commit -m "first commit"

git branch -M main

git remote add origin https://github.com/seu_usuario/seu_projeto.git

git push -u origin main
```

A primeira linha “**echo "# seu_projeto_api" >> README.md**”, informa como criar um arquivo readme, o que pode ser ignorado, pois o Rails cria automaticamente um durante a criação do projeto.

A linha “**git init**”, mostra como iniciar o git no seu projeto, novamente, esse comando pode ser ignorado, pois o Rails também já faz isso automaticamente para você.

Em seguida, o comando “git add README.md” mostra como inserir o arquivo readme na lista de arquivos a serem enviados ao Github, vamos substituir ele por um comando que vai preparar todos os arquivos de uma vez para o envio. Faremos isso com o comando a seguir.

$ git add .

O próximo comando, “ **git commit -m "first commit"** ”, cria uma mensagem informativa sobre o que esta sendo registrado no commit que será feito, você pode alterar ele para o seguinte:

$ git commit -m "1° commit – aplicação criada"

O comando “**git branch -M main**”, informa o branch que será usado, no caso, main.

O comando “**git remote add origin https://github.com/seu_usuario/seu_projeto.git**”, adiciona o endereço de seu projeto no Github.

E finalmente, o comando “**git push -u origin main**”, envia seu projeto para o Github. Siga os passos explicados e seu projeto será mandado para o Github. Vamos seguir com algumas configurações simples agora.

- Configurações do projeto

Agora, vamos fazer algumas pequenas configurações no projeto e instalar alguns recursos. Nesta seção faremos as tarefas listadas:

- Configurar o README.md.

- Instalar as gems necessárias.

- Configurar o rspec.

- Configurar o cors.

- Fazer um commit da aplicação com as alterações feitas.

Para configurar o README.md, basta abrir ele no seu editor de texto e incluir as linhas abaixo e salvar em seguida:

API "consoles_api", desenvolvida em Rails 7, com TDD e versionamento de recursos.

Endpoints da API

-GET: localhost/api/v1/consoles

-GET: localhost/api/v1/consoles/id

-POST: localhost/api/v1/consoles

-PATCH: localhost/api/v1/consoles/id

-DELETE: localhost/api/v1/consoles/id

Com o README editado, insira as linhas abixo no arquivo Gemfile e execute o comando **bundle** em seguida.

```
# Gems do projeto
gem 'rspec-rails'
gem 'rspec-json_expectations'
gem 'rack-cors', require: 'rack/cors'
```

Agora vamos configurar o **rspec**, para isso, primeiramente, execute o comando abaixo.

```
$ rails g rspec:install
```

Em seguida, adicione a linha abaixo ao arquivo **.rspec**, para permitir uma saída mais detalhada na execução dos testes, bem como para gerar um arquivo html com o registro dos resultados dos testes.

```
--format documentation --format html --out spec/resultados_rspec.html
```

Agora, adicione as linhas abaixo ao arquivo **config/application.rb**

```
# Configuração do RSPEC
  config.generators do |g|
    g.test_framework :rspec,
      fixtures: false,
      view_specs: false,
      helper_specs: false,
      routing_specs: false
  end
```

Com o rspec configurado, vamos configurar o **cors** da aplicação adicionando o conteúdo abaixo ao arquivo **config/initilizers/cors.rb**. Lembrando que **cors**, é uma implementação de segurança que permite controlar quem pode acessar a API, e evitar abusos e alguns tipos de ataques. E mesmo em ambiente de desenvolvimento, por exemplo, você pode ter dificuldades de acessar a API no REACT caso não configure o **cors**.

```
Rails.application.config.middleware.insert_before 0, Rack::Cors do
  allow do
    origins 'http://localhost:4000' # endereço que pode fazer requisições a API
```

```
    resource '*', headers: :any, # recursos permitidos
    methods: [:get, :post, :put, :patch, :delete, :options, :head] # métodos http permitidos
  end
end
```

Agora nossa API esta configurada e basta enviar as alterações para o Github, para prosseguir com o desenvolvimento do nosso projeto. Então, digite os comandos abaixo para enviar o projeto para o Github.

$ git add .

$ git commit -m “2° commit – aplicação configurada”

$ git push -u origin main

- Testes de diretórios

Agora, vamos realizar alguns testes para verificar a existência das pastas da nossa estrutura de projeto, lembrando que vamos versionar nossa API, e do modo que precisaremos de algumas pastas para isso. Existem modos diferentes de versionar uma API? Sim, e você pode pesquisar sobre isso no **google**, mas vamos focar na forma que pode ser possivelmente a mais simples. Eu sempre devo testar antes de criar pastas? Depende, do seu estilo de programação bem como, depende da equipe com a qual você vai trabalhar, a forma apresentada neste tutorial **não** é a verdade absoluta, esteja pronto para aprender e se adaptar.

Então, agora crie dentro da sua pasta **spec**, o arquivo **folders_spec.rb**, onde serão inseridos 4 testes, sendo:

- teste de verificação de existência da pasta **spec/models**.

- teste de verificação de existência da pasta **spec/controllers**.

- teste de verificação de existência da pasta **app/controllers/api**.

- teste de verificação de existência da pasta **app/controllers/api/v1**.

Dito isso, inclua no arquivo **spec/folders_spec.rb** o conteúdo a seguir e salve.

```
RSpec.describe 'testes de verificação de existencia de pastas' do # descrição do grupo de testes

  it 'a pasta spec/models existe?' do # descrição deste teste específico
    # espera que a pasta spec/models exista
    expect(Dir.exist?('spec/models')).to eq(true)
  end

end
```

Agora rode o teste com o comando **rspec**, como mostrado a seguir, lembrando que de acordo com o ciclo TDD, este teste vai falhar, pois a pasta '**spec/models**' não existe ainda.

$ rspec

Ótimo, o teste falhou, vamos agora enviar as alterações do projeto para o Github com os comandos a seguir. Atentando para o fato de que você deve inserir uma mensagem descritiva das alterações feitas.

$ git add .

$ git commit -m "3° commit – tdd folders: a pasta spec/models existe? RED**"**

$ git push -u origin main

Em seguida, vamos fazer o teste passar, criando a pasta "**spec/models**". Então, crie a pasta e rode o **rspec** novamente. Desta vez o teste passa, ou seja, a etapa dois do ciclo TDD foi feita.

$ rspec

Agora, basta enviar as alterações para o Github.

$ git add .

$ git commit -m "4° commit – tdd folders: a pasta spec/models existe? GREEN**"**

$ git push -u origin main

Atenção: Deste ponto em diante, para evitar o seu cansaço desnecessário lendo comandos repetitivos, vamos focar em escrever os testes. Pois os passos para fazer os commits no Github são os mesmos, mudando apenas a mensagem descritiva.

Vamos agora adicionar mais um teste, desta vez, vamos testar se a pasta "**spec/controllers**" existe no projeto. Insira o código a seguir no arquivo "**spec/folders_spec.rb**".

```
it 'a pasta spec/controllers existe?' do # descrição deste teste específico
   # espera que a pasta spec/controllers exista
   expect(Dir.exist?('spec/controllers')).to eq(true)
  end
```

Rode o comando **rspec** e verá o teste adicionado falhar, como esperado, pois a pasta testada não existe ainda. Em seguida, envie as alterações para o Github.

Agora, vamos fazer o teste passar, criando a pasta "**spec/controllers**". Crie a pasta e rode o **rspec** novamente, desta vez o teste passa e mais uma vez o passo dois do ciclo TDD foi feito. Envie as alterações para o Github e vamos para outro teste.

Vamos testar se a pasta "**app/controllers/api**", usada para versionarmos nossa API, esta presente no projeto. Faremos isso com o teste a seguir.

```
it 'a pasta app/controllers/api existe?' do # descrição deste teste específico
   # espera que a pasta  app/controllers/api exista
   expect(Dir.exist?('app/controllers/api')).to eq(true)
end
```

Execute o comando **rspec**, e veja o teste adicionado agora falhar, tal como esperado. Submeta as alterações ao Github.

Vamos fazer o teste passar agora, criando a pasta necessária, "**app/controllers/api**". Crie a pasta, rode o **rspec** novamente e você verá o teste passar. Mais uma vez, faça o commit das alterações.

Faremos agora o último teste desta bateria, vamos verificar a existência da pasta "**app/controllers/api/v1**", para isso, insira o teste a seguir no arquivo "**spec/controllers**". Salve o arquivo, rode o **rspec** e veja o teste falhar. Em seguida faça o commit das alterações.

```
it 'a pasta app/controllers/api/v1 existe?' do # descrição deste teste específico
    # espera que a pasta  app/controllers/api/v1 exista
    expect(Dir.exist?('app/controllers/api/v1')).to eq(true)
end
```

Faremos agora o último teste inserido passar, criando a pasta "**app/controllers/api/v1**" e rodando o **rspec** novamente. O teste ira passar, em seguida, faça o commit das alterações.

Pronto, nossa primeira bateria de testes foi finalizada e o seu arquivo "spec/folders_spec.rb" deve ficar como o exemplo a seguir.

```
RSpec.describe 'testes de verificação de existencia de pastas' do # descrição do grupo de testes


  it 'a pasta spec/models existe?' do # descrição deste teste específico
    # espera que a pasta spec/models exista
    expect(Dir.exist?('spec/models')).to eq(true)
  end

  it 'a pasta spec/controllers existe?' do # descrição deste teste específico
    # espera que a pasta spec/controllers exista
    expect(Dir.exist?('spec/controllers')).to eq(true)
  end

  it 'a pasta app/controllers/api existe?' do # descrição deste teste específico
    # espera que a pasta  app/controllers/api exista
    expect(Dir.exist?('app/controllers/api')).to eq(true)
  end

   it 'a pasta app/controllers/api/v1 existe?' do # descrição deste teste específico
      # espera que a pasta  app/controllers/api/v1 exista
     expect(Dir.exist?('app/controllers/api/v1')).to eq(true)
  end

end
```

Observe que o processo de desenvolvimento com TDD é bem simples, mais um pouco repetitivo, lembre que fizemos a sequência de passos a seguir, e tenha em mente que ela será usada em todo nosso projeto.

- 1° criar um teste que falha e fazer commit.

- 2° fazer o teste passar e fazer commit.

Vamos para a próxima bateria de testes.

- Testes de model

Devemos testar models? Novamente, depende do seu estilo de programação, do projeto, do tempo, da equipe e do que já está estabelecido, enfim, esteja disposto a se adaptar caso necessário.

Aqui neste tutorial, vamos fazer esses testes devido ao objetivo deste tutorial, ser uma **introdução**.

Vamos iniciar nossa nova bateria de testes agora, crie o arquivo "**spec/models/console_spec.rb**", em seguida, insira nele o código a seguir referente ao nosso primeiro teste. Esse primeiro teste é muito simples, e apenas irá testar se o campo **name** consegue ser preenchido. Esse primeiro teste irá falhar, de modo esperado, pois ainda não temos um model criado e nem uma estrutura no banco de dados que possa ser utilizada. E esse é um dos pontos fortes do TDD, ele orienta você sobre as partes que precisam ser incluídas no seu código.

```
require 'rails_helper'

RSpec.describe Console, type: :model do

  before{@console = Console.new} # objeto console para uso nos testes

  describe ' Testes de preenchimento de campos de um objeto console ' do

    it 'name consegue ser preenchido?' do
      @console.name = ' '
      expect(@console.name).to eq('ps5') # espera que name seja igual a ps5
    end

  end

end
```

Rode o comando **rspec** e veja o teste falhar, em seguida, envie a alteração para o Github.

Para fazer o teste passar, vamos precisar primeiramente criar um model e em seguida rodar uma migração. Para então, preencher de fato o valor do campo **name** e ver o teste passar. Então, agora, execute os comandos a seguir.

$ rails g model Console name manufacturer # gera o model Console com os campos name e manufacturer

$ rails db:migrate # configura o banco de dados de acordo com o definido no comando anterior

Agora, insira um nome para o campo **name** e rode o **rspec** novamente, você verá o teste passar. Seu teste deverá fica como a seguir.

```
it 'name consegue ser preenchido?' do
  @console.name = 'ps5'
  expect(@console.name).to eq('ps5') # espera que name seja igual a ps5
end
```

Feito o teste passar, faça o commit.

Vamos agora testar o preenchimento do campo **manufacturer**, o teste será como o que segue. Rode o **rspec** e veja o teste falhar, em seguida, envie a alteração para o Github.

```
it 'manufacturer consegue ser preenchido?' do
  @console.manufacturer = ' '
  expect(@console.manufacturer).to eq('sony') # espera que manufacturer seja igual a  sony
end
```

Para fazer o teste passar, basta preencher um valor para o campo **manufacturer**. Seu código ficará como o que segue. Rode o **rspec** e veja o teste passar em seguida, envie a alteração para o Github.

```
it 'manufacturer consegue ser preenchido?' do
  @console.manufacturer = 'sony'
  expect(@console.manufacturer).to eq('sony') # espera que manufacturer seja igual a sony
end
```

Finalizada esta etapa, vamos fazer alguns testes de validação de um objeto console. Para isso, insira as instruções a seguir no arquivo "**spec/models/console_spec.rb**", abaixo do último teste.

```
describe 'Testes de validação de campos de um objeto console' do

  it 'objeto console valido com campos obrigatórios preenchidos?' do
    @console.name = ' '
    @console.manufacturer = ' '
    expect(@console).to be_valid
  end

end
```

Agora, no arquivo "**app/models/console.rb**", insira a linha abaixo para habilitar a validação dos campos **name** e **manufacturer**.

validates :name, :manufacturer, presence: true

Agora, rode o **rspec** e veja o teste falhar e observe que será exibida a mensagem de erro "**Name can't be blank, Manufacturer can't be blank**", informando que os campos não podem ficar em branco, de acordo com o que foi definido no model. Faça agora o commit das alterações.

Para fazer o teste passar, vamos preencher os campos **name** e **manufacturer**, seu código deve ficar como a seguir.

```
it 'objeto console valido com campos obrigatórios preenchidos?' do
  @console.name = 'xbox'
  @console.manufacturer = 'microsoft'
  expect(@console).to be_valid # espera que objeto console com campos preenchidos seja valido
end
```

Execute o **rspec** e veja o teste passar, em seguida, faça o commit das alterações.

Vamos finalizar esta seção adicionando um teste que verifica se um objeto **console** é invalido com os campos de preenchimento obrigatório em branco, seu código pode ser como a seguir.

```
it 'objeto console invalido com campos obrigatórios não preenchidos?' do
  console = Console.new # campos name e manufacturer não preenchidos
  expect(console).to be_valid
end
```

Observe que a lógica aqui foi invertida no expect, mas existem outras formas de fazer essa verificação. Rode o comando **rspec**, veja o teste falhar e em seguida, faça o commit das alterações.

Finalmente, para fazer o teste passar, basta trocarmos a função “**be_valid**” por “**be_invalid**”.

Em seguida, rode o **rspec** e veja o teste passar, logo após, faça o commit das alterações. Seu código final no arquivo “**spec/models/console_spec.rb**” deve ficar como o exemplo que segue.

```
require 'rails_helper'

RSpec.describe Console, type: :model do

  before{@console = Console.new} # objeto console para uso nos testes

  describe 'Testes de preenchimento de campos de um objeto console' do

    it 'name consegue ser preenchido?' do
      @console.name = 'ps5'
      expect(@console.name).to eq('ps5') # espera que name seja igual a ps5
    end

    it 'manufacturer consegue ser preenchido?' do
      @console.manufacturer = 'sony'
      expect(@console.manufacturer).to eq('sony') # espera que manufacturer seja igual   a sony
    end

  end

  describe 'Testes de validação de campos de um objeto console' do

    it 'objeto console valido com campos obrigatórios preenchidos?' do
      @console.name = 'xbox'
      @console.manufacturer = 'microsoft'
      expect(@console).to be_valid
    end

    it 'objeto console invalido com campos obrigatórios não preenchidos?' do
      console = Console.new # campos name e manufacturer não preenchidos
      expect(console).to be_valid
    end

  end

end
```

Terminamos nossa bateria de testes de model e agora falta apenas a última, e veremos ela a seguir.

- Testes de controller

Agora, vamos finalizar nossa jornada de testes, realizando os testes de controller. E novamente uma pergunta, eu devo testar controllers? Depende, de como você quer programar e de como o projeto que você vai atuar está definido, depende de como as pessoas no projeto preferem realizar os testes.

Em todo caso, conheça a mecânica de como testar e esteja pronto pra se adaptar quando necessário.

Agora, crie o arquivo "**spec/controllers/consoles_controller_v1_spec.rb**". Esse arquivo deverá ter inicialmente o código abaixo, acrescente-o. O teste tem o objetivo de listar todos os registros salvos no banco.

```
require 'rails_helper'

RSpec.describe Api::V1::ConsolesController, type: :controller do

  # objeto usado durante alguns testes
  before{@console = Console.create(name: 'snes', manufacturer: 'nintendo')}

  # descreve que este teste irá realizar um get no endpoint /api/v1/consoles
  describe 'GET /api/v1/consoles' do
    it 'Consegue listar todos os consoles e retornar status 200?' do
    get :index # faz uma requisição get para o controller

    # faz um parse da response para json e espera que seu tamanho seja 1
    expect(JSON.parse(response.body).size).to eq(1)
  end
end

end
```

Rode o **rspec** e veja o teste falhar, pois ainda não temos nem o controller e nem a rota necessários. Em seguida, faça o commit das alterações.

Para fazer o teste passar, vamos:

1° criar o controller "**Api::V1::ConsolesController**".

2° configurar o arquivo "**config/routes.rb**"

Observe que o controller ficará na pasta "**app/controllers/api/v1**", por isso a escrita "**Api::V1::ConsolesController**". Essa é a base do versionamento da nossa API, essa estrutura de pastas. Se quiséssemos criar uma versão 2 da nossa api, bastaria seguir o padrão, trocando **v1** por **v2**, e assim por diante. E configurando as rotas também.

Portanto, para fazer o teste passar agora, crie o arquivo "**app/controllers/api/v1/consoles_controller.rb**", e lembre que ele é um classe Ruby. Inclua nele o código abaixo.

```
# classe herda de ApplicationController
class Api::V1::ConsolesController < ApplicationController

# função usada antes das actions show, update e destroy
before_action :set_console, only: %i[show update destroy]

# retorna todos os consoles cadastrados em formato json
def index
  @consoles = Console.all
  render json: @consoles
end


# funções privadas
private

# busca no banco um console com id enviado na requisição
def set_console
  @console = Console.find(params[:id])
end

# permite que os campos name e manufacturer sejam manipulados nas requisições
def console_params
  params.require(:console).permit(:name, :manufacturer)
end

end
```

Execute o rspec agora, e veja o teste falhar, mas desta vez, atente para a mensagem de erro "**No route matches**", indicando que não temos as rotas configuradas na aplicação. Para resolver isso, abra o arquivo "**config/routes**", e inclua o código abaixo.

```
namespace :api do # faz a pasta api ser entendida nas requisições
  namespace :v1 do # faz a pasta v1 ser entendida nas requisições
    resources :consoles # habilita as rotas do CRUD
  end
end
```

Agora, rode o **rspec** e veja os testes passarem. Em seguida, faça o commit das alterações.

Pela primeira vez, vamos fazer o ciclo TDD completo, **red, green, refactor**. Até qui, usamos apenas duas etapas, mas agora vamos usar o refactor, que nada mais é do que uma melhoria no seu teste. No caso do nosso teste, vamos verificar se a resposta que recebemos possui o status 200.

Inclua a instrução abaixo no seu teste.

```
expect(response).to have_http_status(200)
```

Rode o **rspec** novamente e veja o teste passar e em seguida, faça o commit. O seu primeiro teste fica como mostrado abaixo.

```
 # descreve que este teste irá realizar um get no endpoint /api/v1/consoles
  describe 'GET /api/v1/consoles' do
    it 'Consegue listar todos os consoles e retornar status 200?' do
    get :index # faz uma requisição get para o controller

    # faz um parse da response para json e espera que seu tamanho seja 1
    expect(JSON.parse(response.body).size).to eq(1)
    # verifica se a resposta possui status 200
    expect(response).to have_http_status(200)
  end
end
```

Agora, vamos fazer o próximo teste, que consiste em verificar se conseguimos obter um registro específico do banco. Inclua o código a seguir no arquivo de testes e rode o **rspec** para ver o teste falhar. Em seguida, faça commit das mudanças.

```
# descreve que este teste irá realizar um get no endpoint /api/v1/consoles/id
describe 'GET /api/v1/consoles/id' do
  it 'Consegue listar um console especifico e retornar status 200?' do
    # faz um get para o controller enviando o id como parâmetro de busca
    get :show, params: {id: @console.id}
    # espera que no corpo da resposta haja o json snes
    expect(response.body).to include_json(name: 'snes')
  end
end
```

Para fazer o teste passar, crie no arquivo "**app/controllers/api/v1/consoles_controller.rb**" a função **show**, conforme código a seguir. Em seguida, rode o **rspec** e veja o teste passar. Após isso, faça o commit.

```
# retorna um registro específico identificado pelo id na requisição
def show
  render json: @console
end
```

Faça agora o passo 3 do ciclo TDD, **refactor**, no seu teste, acrescentando a linha a seguir. Rode a **rspec** e tudo deve passar, logo após, faça o commit necessário.

```
# verifica se a resposta possui status 200
expect(response).to have_http_status(200)
```

O seu teste ficará como a seguir.

```
# descreve que este teste irá realizar um get no endpoint /api/v1/consoles/id
describe 'GET /api/v1/consoles/id' do
  it 'Consegue listar um console especifico e retornar status 200?' do
    # faz um get para o controller enviando o id como parâmetro de busca
    get :show, params: {id: @console.id}
    # espera que no corpo da resposta haja o json name
    expect(response.body).to include_json(name: 'snes')
    # verifica se a resposta possui status 200
    expect(response).to have_http_status(200)
  end
end
```

Vamos agora ao próximo teste, vamos ver se conseguimos criar um novo registro no banco, para isso, inclua no seu arquivo de testes o código a seguir, salve e rode o **rspec** em seguida, você verá o teste falhar. Pois ainda não temos a funcionalidade implementada.

```
# descreve que um post será feito no endpoint /api/v1/consoles
describe 'POST /api/v1/consoles' do
  it 'Consegue criar um console e retornar status 201?' do
    # faz uma requisição post enviando os dados de name e manufacturer para o controller
    post :create, params: {console: {name: 'mega-drive', manufacturer: 'sega'},  format: :json}
    # espera que no corpo da resposta haja o json name
    expect(response.body).to include_json(name: 'mega-drive')
  end
end
```

Para fazer o teste passar, vamos inserir o código a seguir no arquivo "**app/controllers/api/v1/consoles_controller.rb**", salve o arquivo e em seguida, rode o **rspec** e o teste irá passar.

```
# cria um console no banco
def create
  @console = Console.new(console_params)
  if @console.save
    render json: @console, status: :created, location: api_v1_console_url(@console)
  else
    render json: @console.errors, status: :unprocessable_entity
  end
end
```

Vamos agora refatorar nosso teste, o passo 3 do TDD, para isso, inclua a linha a seguir no teste, salve e rode o **rspec**, tudo continua passando. Faça o commit das alterações.

```
expect(response).to have_http_status(201)
```

Seu teste completo deve ficar como o código a seguir.

```ruby
# descreve que um post será feito no endpoint /api/v1/consoles
describe 'POST /api/v1/consoles' do
  it 'Consegue criar um console e retornar status 201?' do
    # faz uma requisição post enviando os dados de name e manufacturer para o controller
    post :create, params: {console: {name: 'mega-drive', manufacturer: 'sega'},  format: :json}
    # espera que no corpo da resposta haja o json name
    expect(response.body).to include_json(name: 'mega-drive')
    # espera que a resposta tenha o status 201
    expect(response).to have_http_status(201)
  end
end
```

Agora, vamos fazer o teste que verifica se conseguimos realizar a atualização de um registro no banco, para isso, inclua o código a seguir no seu arquivo de teste de controllers, salve e rode o **rspec** para ver seu teste falhar como esperado. Em seguida, faça o commit das mudanças.

```ruby
# descreve um patch que será feito no endpoint /api/v1/consoles/id
describe 'PATCH /api/v1/consoles/id' do
  it 'Consegue atualizar um console retornar status 200?' do
    # busca no banco o último registro, o que será atualizado
    console = Console.last
    # solicita a atualização do registro enviando os dados name e manufacturer
    patch :update, params: {console: {name: 'dreamcast', manufacturer: 'sega'}, id: console.id}
    #  espera que no corpo da resposta haja o json name
    expect(response.body).to include_json(name: 'dreamcast')
  end
end
```

Para fazer o teste de atualização passar, vamos inserir o código a seguir no controller. Em seguida, salve o arquivo, rode o **rspec** e você verá o teste passar. Lembre de fazer o commit necessário.

```ruby
# atualiza um console no banco
def update
  if @console.update(console_params)
    render json: @console
  else
    render json: @console.errors, status: :unprocessable_entity
  end
end
```

Para finalizar o teste, vamos refatorar. Para isso, inclua a linha a seguir no teste em questão, salve e rode o **rspec**, tudo vai continuar passando. Faça o commit necessário.

```
expect(response).to have_http_status(200)
```

Seu teste completo deve ficar como o código a seguir.

```
# descreve um patch que será feito no endpoint /api/v1/consoles/id
describe 'PATCH /api/v1/consoles/id' do
  it 'Consegue atualizar um console retornar status 200?' do
    # busca no banco o último registro, o que será atualizado
    console = Console.last
    # solicita a atualização do registro enviando os dados name e manufacturer
    patch :update, params: {console: {name: 'dreamcast', manufacturer: 'sega'}, id: console.id}
    # espera que no corpo da resposta haja o json name
    expect(response.body).to include_json(name: 'dreamcast')
    # espera que a resposta tenha o status 200
    expect(response).to have_http_status(200)
  end
end
```

Finalmente chegamos ao último teste, que consiste em verificar se conseguimos excluir um registro do banco, para isso, vamos inserir no arquivo de testes de controllers, o código a seguir.

Após inserir, salve o arquivo, e rode o **rspec** para ver o teste falhar. Em seguida, faça o commit das alterações.

```
# descreve que um delete será feito no endpoint /api/v1/consoles/id

describe 'DELETE /api/v1/consoles/id' do
  it 'Consegue excluir um console e retornar status 204?' do
    # busca no banco o último registro, o que será apagado
    console = Console.last
    # solicita a exclusão do registro enviando o id do registro a excluir
    delete :destroy, params: {id: console.id}
    # ao buscar todos os consoles no banco, espera que o registro excluído não seja encontrado
    expect(Console.all).not_to include(console)
  end
end
```

Para fazer o teste passar, vamos inserir no controller, o código a seguir. E após inserir, salve o arquivo e rode **rspec** para ver o teste passar. Lembre de fazer o commit necessário.

```
# exclui um registro do banco
def destroy
  @console.destroy!
end
```

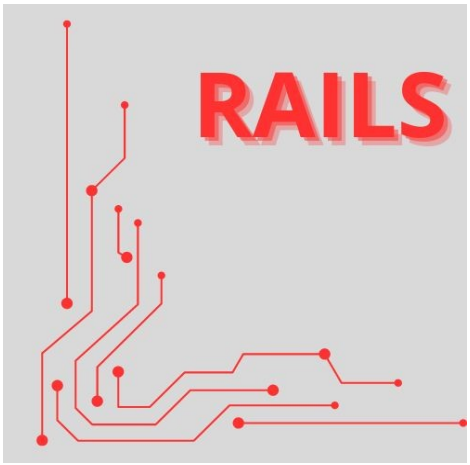
Vamos refatorar o teste incluindo a linha abaixo, após inserir, salve o arquivo, rode o **rspec**, e tudo continua passando.

```
expect(response).to have_http_status(204)
```

O teste completo fica como o código abaixo.

```
# descreve que um delete será feito no endpoint /api/v1/consoles/id

describe 'DELETE /api/v1/consoles/id' do
  it 'Consegue excluir um console e retornar status 204?' do
    # busca no banco o último registro, o que será apagado
    console = Console.last
    # solicita a exclusão do registro enviando o id do registro a excluir
    delete :destroy, params: {id: console.id}
    # ao buscar todos os consoles no banco, espera que o registro excluído não seja encontrado
    expect(Console.all).not_to include(console)
    # espera que a resposta tenha o status 204
    expect(response).to have_http_status(204)
  end
end
```

Neste ponto, concluímos nossa pequena API, e ela está pronta para receber requisições via Postman, navegador Web, ou qualquer framework Frontend como React e outros. Vamos ao próximo capítulo.



Capítulo 11

Testando a API com o Postman

Para testar nossa API, vamos utilizar o Postman, faremos testes nos nossos 5 endpoints, como esquema a seguir.

GET /api/v1/consoles # listar todos os registros

GET /api/v1/consoles/id # listar um registro específico buscando pelo id

POST /api/v1/consoles # criar um registro

PATCH /api/v1/consoles/id # atualizar um registro buscando pelo id

DELETE /api/v1/consoles/id # apagar um registro pelo id

Primeiramente, vamos configurar nosso arquivo "**db/seeds.rb**", para popular nosso banco com alguns registros. Então, insira o código a seguir no seu **seeds.rb** e salve o arquivo, em seguida faça o commit das alterações.

```
puts 'gerando consoles…'
  # executa 5 vezes a instrução para criar registros no banco
  5.times do
    # cria registros no banco com valores selecionados dentro da lista disponível
    Console.create(name: ['ps3', 'ps4', 'ps5'].sample, manufacturer: 'sony')
  end
puts 'consoles gerados com sucesso'
```

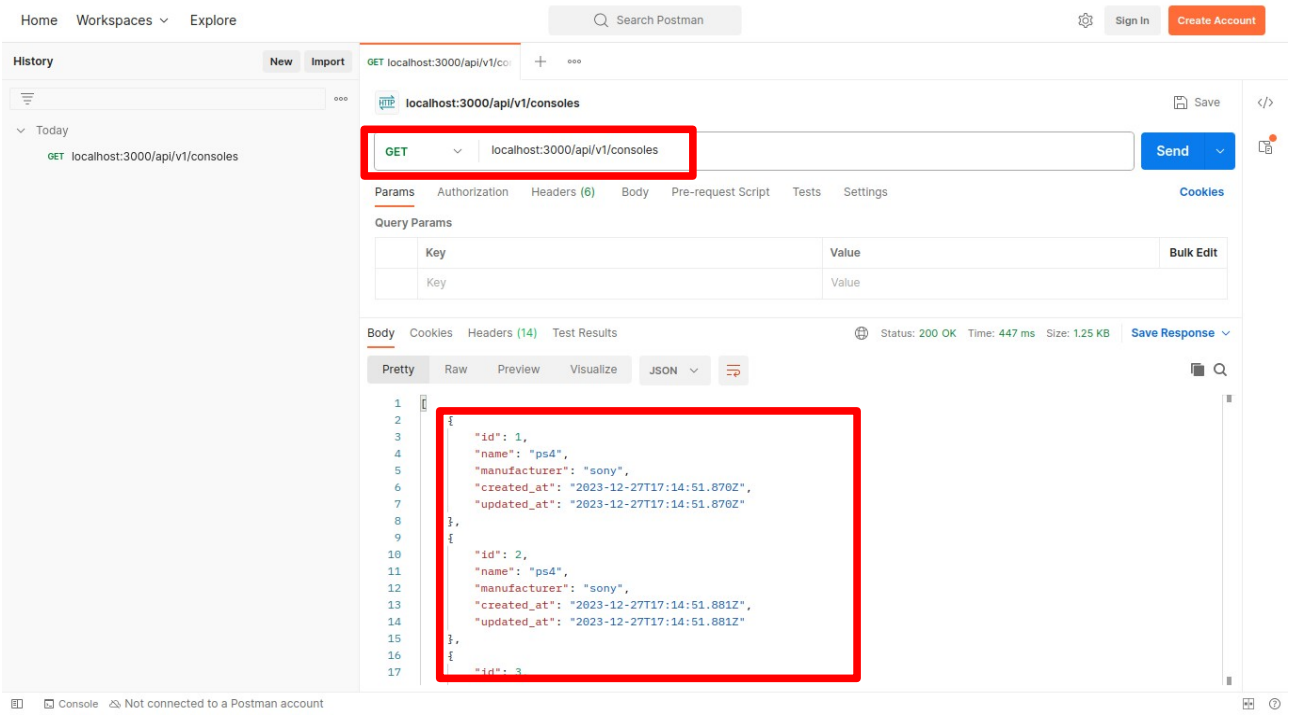
Agora, execute o comando abaixo para: **apagar**, **criar**, **migrar** e **popular**, o banco de dados. Sim, tudo com um comando. "**Facas afiadas**", lembra?

$ **rails db:drop db:create db:migrate db:seed**

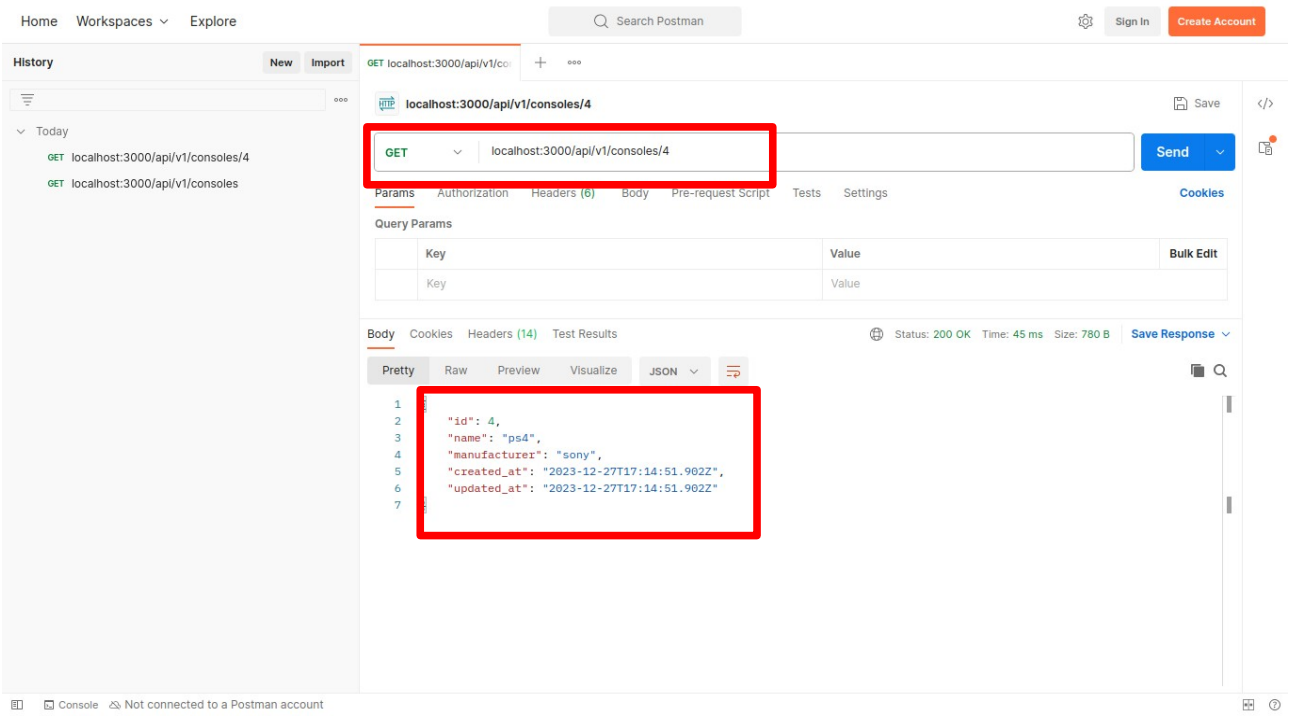
Em seguida, inicie a aplicação com o comando a seguir para deixar ela ouvindo as requisições que faremos a seguir.

$ rails s

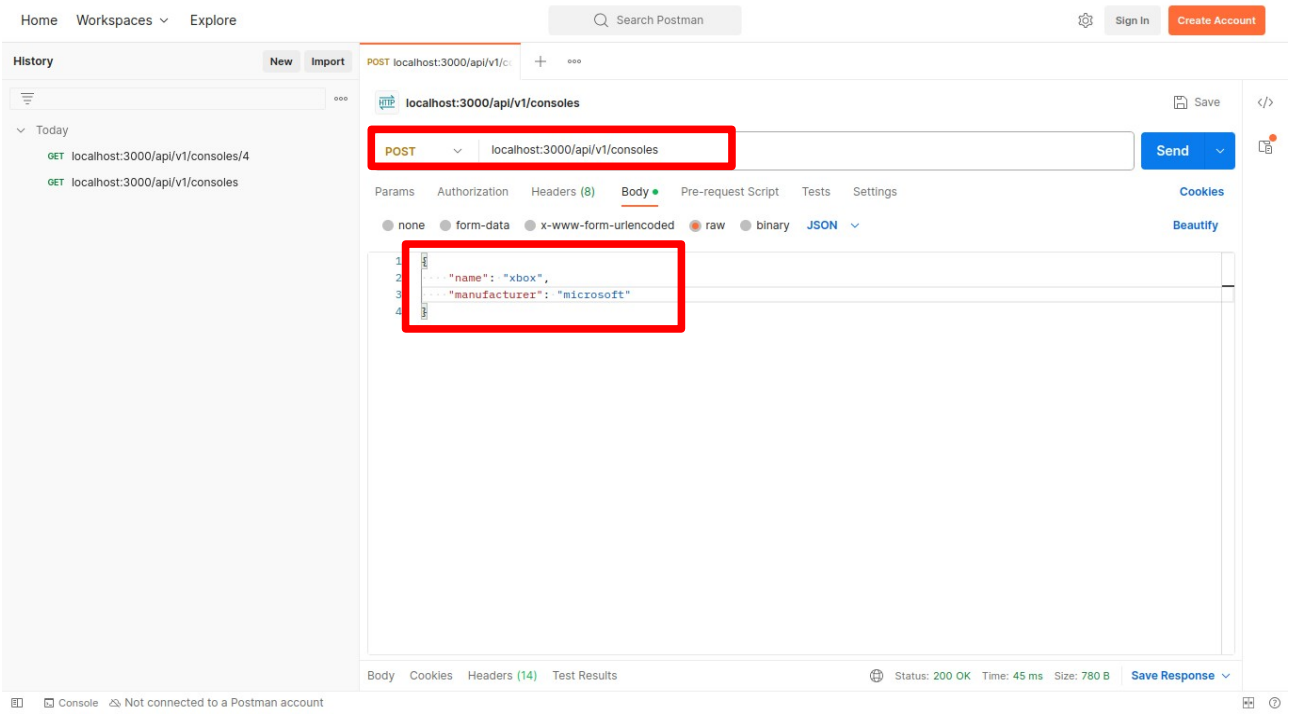
Nosso primeiro teste será uma requisição **GET** através do Postman, para listar todos os registros. Como mostra a imagem abaixo.



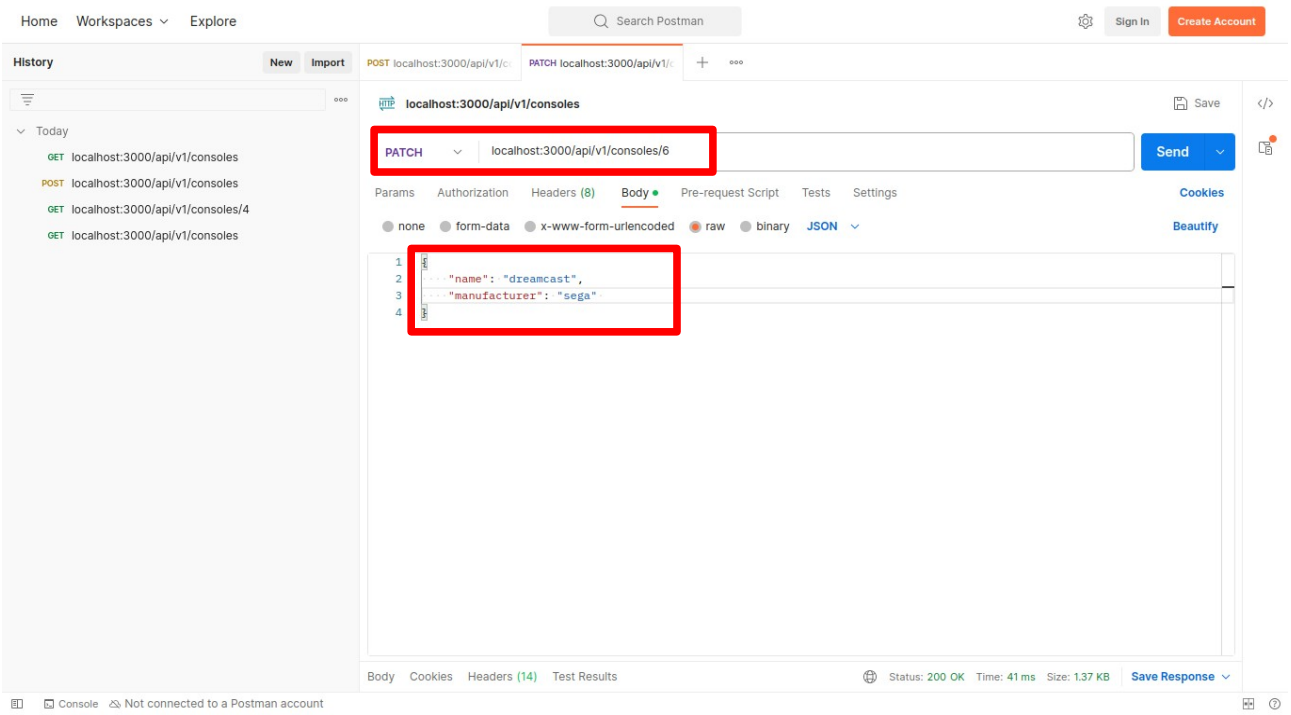
Nosso segundo teste é uma requisição **GET** para listar um registro do banco em específico.



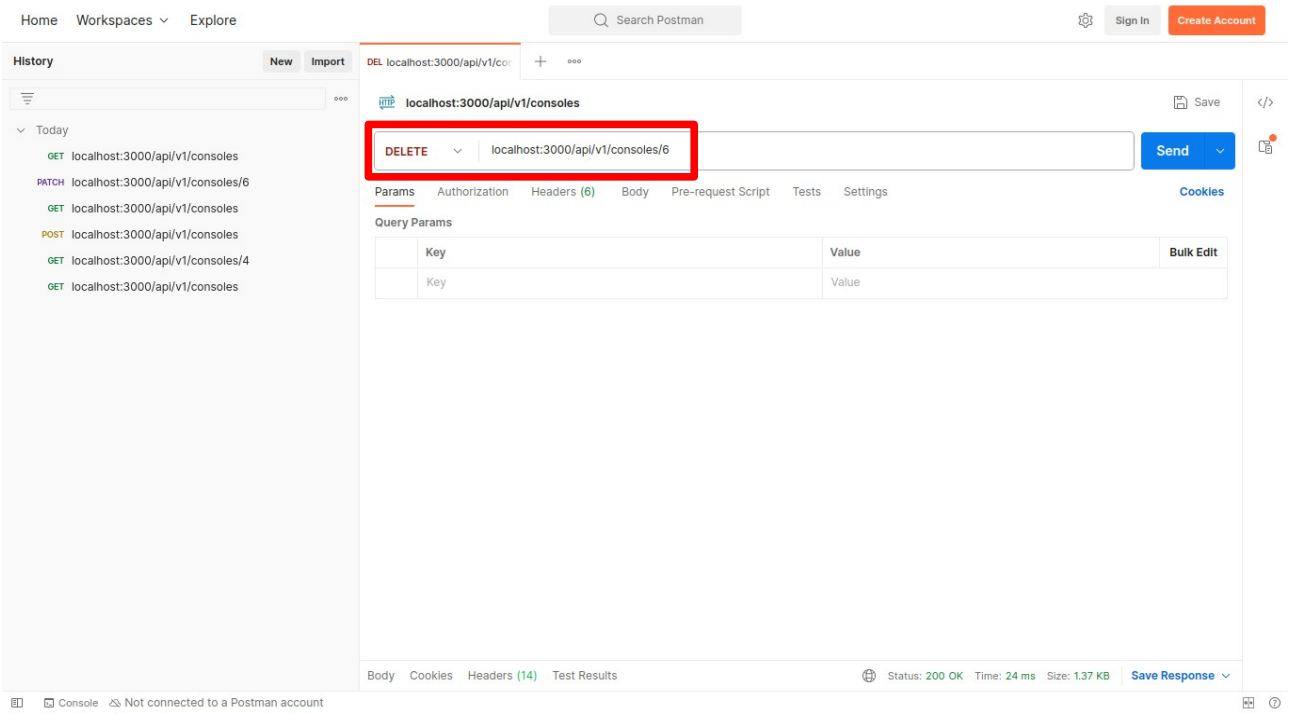
Nosso terceiro teste será uma requisição **POST**, para criar um registro no banco, como imagem a seguir.



Nosso quarto teste é uma requisição **PATCH** para atualizar um registro do banco. Como a imagem a seguir.



Nosso quinto teste é uma requisição **DELETE**, para excluir um registro do banco. Como mostra a imagem a seguir.



E finalizamos nossa API, caro leitor, este tutorial não é uma referência no assunto, é uma introdução. Logo, é altamente recomendável que você busque se aprimorar, estudando e escrevendo código. Você já deu um passo importante, continue assim. Sucesso.

Agradecimentos

Agradeço primeiramente a Deus. E agradeço aos meus pais que sempre me ajudaram. Agradeço ao leitor. E agradeço ao professor **Jackson Pires**, com o qual através de seus cursos, aprendi sobre Ruby e Rails.

Referências

Beginning Ruby3, Carleton DiLeo, Peter Cooper, Fourth Edition

Ruby on Rails Tutorial, Michael Hartl, Fourth Edition

Cursos do professor Jackson Pires