



Plataformas de aplica  es Web

Aula 06 - Plataformas back-end



Material Did tico do Instituto Metr pole Digital - IMD

Termo de uso

Os materiais did ticos aqui disponibilizados est o licenciados atrav s de Creative Commons **Atribui  o-SemDeriva  es-SemDerivados CC BY-NC-ND**. Voc  possui a permiss o para realizar o download e compartilhar, desde que atribua os cr ditos do autor. N o poder  alter -los e nem utiliza-los para fins comerciais.

Atribui  o-SemDeriva  es-SemDerivados

CC BY-NC-ND



<https://creativecommons.org/licenses/by-nc-nd/4.0/>

Apresentação

As plataformas de desenvolvimento web que fornecem facilidades para se criar aplicações do lado do servidor são chamadas plataformas back-end ou frameworks backend.

Elas têm diferentes tipos de complexidade e funcionalidades. Algumas fornecem somente uma maneira de gerenciar rotas e executar ações no servidor, outras já vem com algum mecanismo de acesso a banco de dados, outras são um pouco mais completas, com mais funcionalidades que as deixam próximas de plataformas full-stack

Nessa aula vamos conhecer algumas plataformas back-end, suas características e linguagem de programação que utilizam.

Exemplos de plataformas back-end

Nessa disciplina, definimos como plataformas back-end os frameworks que são mais simples que as full-stack pois normalmente fornecem um conjunto de funcionalidade mais enxuto, mas com possibilidade de expansão.

É sim possível criar uma aplicação completa utilizando somente uma plataforma back-end, porém quase sempre será necessário adicionar bibliotecas extras para ajudar na renderização das páginas HTML, acessar bancos de dados, autenticar usuários, tratar upload de arquivos etc. Diferente das plataformas full-stack, as plataformas back-end normalmente não trazem junto essas funcionalidades (com algumas exceções) e cabe ao programador(a) adicionar os módulos que desejar.

Alguns exemplos de plataformas back-end são

- ExpressJS (Javascript) - <https://expressjs.com/>
- Sinatra (Ruby) - <http://sinatrarb.com/>
- Spring Boot (Java) - <https://spring.io/projects/spring-boot>
- Flask (Python) - <https://flask.palletsprojects.com/>
- Slim (PHP) - <https://www.slimframework.com/>

Plataformas back-end mais enxutas são excelentes pontos de partida para a criação de aplicações com ações que rodam no servidor. Por oferecerem menos funcionalidades prontas, elas tem uso indicado em situações onde o programador(a) deseja ter maior controle de que bibliotecas extras deseja utilizar. Isso adiciona mais flexibilidade mas também demanda que o programador(a) tenha um trabalho extra juntando diversas tecnologias. Por exemplo, o Express, que utiliza javascript, não vem com um mecanismo para tratar upload de arquivos, mas você pode instalar o *multer*, uma biblioteca que trata dessa funcionalidade.

Vantagens das plataformas back-end

As plataformas back-end, assim como as full-stack, também existem ao redor de uma linguagem de programação específica e cabe ao programador(a) que a deseja utilizar, conhecer essa linguagem minimamente para que possa a utilizar.

Alguns exemplos são o ExpressJS (Javascript), Sinatra(Ruby), Spring Boot(Java), Flask(Python) e Slim(PHP).

Algumas dessas plataformas também se denominam micro-frameworks back-end, ou seja, frameworks mínimos para que você comece a desenvolver sua solução.

A vantagem de se utilizar uma dessas plataformas no seu sistema é o fato delas não terem praticamente nenhuma opinião com relação a como deve ser o seu sistema e oferecem somente o básico para se ter um sistema web que é capaz de receber requisições, rodar alguma ação e retornar um resultado. Utilizando essas plataformas, você é quem deve se preocupar com questões de segurança, autenticação, autorização, acesso a dados, geração de HTML dinâmico, etc.

Desvantagens das plataformas back-end

A desvantagem de utilizar plataformas assim é que o programador(a) necessita ter mais trabalho para se iniciar a sua aplicação e também ele precisa ser capaz de definir como será a arquitetura da solução em mais detalhes e sem nenhuma interferência. Alguns podem ver isso como vantagem e desvantagem, dependendo da sua experiência.

Plataformas back-end normalmente são utilizadas em conjunto com outras tecnologias para se obter um sistema útil e completo na prática.

É importante observar que você pode utilizar uma plataforma full-stack somente para a parte back-end do seu sistema, bastando não utilizar todos os recursos da plataforma full-stack, tornando-a, efetivamente, uma plataforma back-end. Por exemplo, você pode utilizar um framework como o RubyOnRails ou o Laravel (ambos definitivamente full-stack) para criar uma simples API back-end, sem nenhuma funcionalidade extra, o que funciona sim, mas você estará utilizando nesses casos frameworks mais pesados para fazer uma aplicação simples, o que pode tornar sua aplicação mais "pesada" na hora de ser executada do que o que seria necessário. Realmente existem interseções de definição aqui.

Vamos a seguir conhecer melhor algumas plataformas back-end.

Express (Javascript)

Web site: <https://expressjs.com/>

Os desenvolvedores do Express definem a plataforma como "Um framework web rápido, sem opinião e minimalista para Node.js"

The screenshot shows the Express.js website homepage. At the top, there is a navigation bar with the 'Express' logo, a search bar, and links for 'Home', 'Getting started', 'Guide', 'API reference', 'Advanced topics', and 'Resources'. The main heading is 'Express 4.17.1', followed by the tagline 'Fast, unopinionated, minimalist web framework for Node.js'. Below this is a terminal snippet: '\$ npm install express --save'. A yellow banner announces 'Express 5.0 alpha documentation is now available.' with a link to the alpha API documentation and a link to the release history. The page is divided into three columns: 'Web Applications' (describing Express as a minimal and flexible Node.js web application framework), 'APIs' (describing the HTTP utility methods and middleware), and 'Performance' (describing the thin layer of fundamental web application features).

Website do Express. Fonte: <https://expressjs.com/>

O Node.js é um runtime Javascript, ou seja, ele permite executar programas escritos em Javascript do lado do servidor e não no navegador. Isso possibilita a criação de aplicações back-end e full-stack utilizando a linguagem Javascript, que normalmente era utilizada somente para códigos que rodavam no navegador.

O Express é um framework web para Node.js (runtime Javascript do lado do servidor) extremamente popular com a comunidade de desenvolvedores da linguagem de programação Javascript.

É possível se criar uma aplicação que processa requisições HTTP no lado do servidor utilizando somente o Javascript no Node.js, sem a necessidade de nenhum framework. Mas com frameworks como o Express essa tarefa se torna mais simples e sem adicionar muita perda de desempenho.

Exemplo de uma aplicação Node.js sem o Express

Abaixo veja uma aplicação que é um servidor HTTP utilizando somente Javascript sem nenhum framework e que implementa uma simples API que responde a dois "endpoints", um para retornar um JSON com uma mensagem de saudação e outro que retorna a hora atual. Você vai precisar instalar o Node.js na sua máquina se desejar executar esses exemplos (<https://nodejs.org/>).

```
const http = require("http");
const PORT = process.env.PORT || 5000;

const server = http.createServer(async (req, res) => {
  //set the request route
  if (req.url === "/ola_servidor" && req.method === "GET") {
    //response headers
    res.writeHead(200, { "Content-Type": "application/json" });
    //set the response
    res.write(JSON.stringify({
      message: "Oi coleguinhas, tudo tranquilo com vocês?"
    }));
  };
  //end the response
  res.end();
  return;
}

  if (req.url === "/que_horas_sao_por_favor" && req.method ===
"GET") {
    //response headers
    res.writeHead(200, { "Content-Type": "application/json" });
    //set the response
    const dataHora = new Date().toLocaleString();
    res.write(JSON.stringify({
      horaAtual: dataHora
    }));
    //end the response
    res.end();
    return;
  }

  // If no route present
  res.writeHead(404, {
```

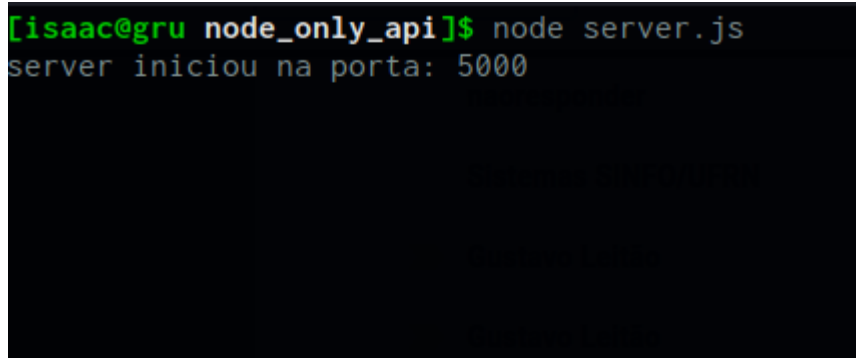
```
    "Content-Type": "application/json" }
  );
  res.end(JSON.stringify({
    message: "Rota não encontrada"
  }));
});

server.listen(PORT, () => {
  console.log(`server iniciou na porta: ${PORT}`);
});
```

Esse exemplo com Javascript puro acima pode ser colocado em um arquivo chamado `server.js` e executado com o comando:

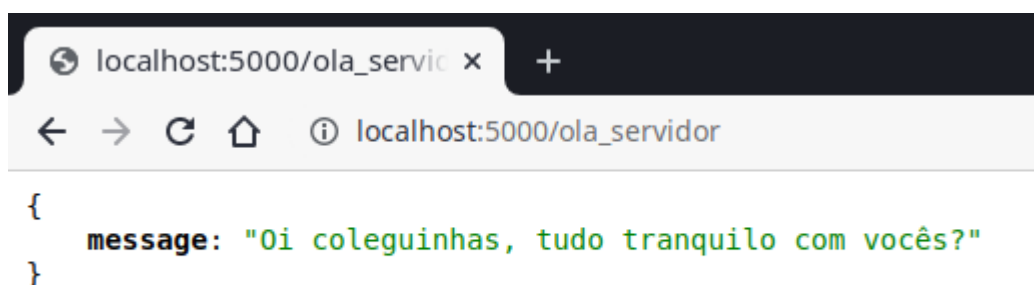
```
node server.js
```

Você verá a seguinte saída no terminal:

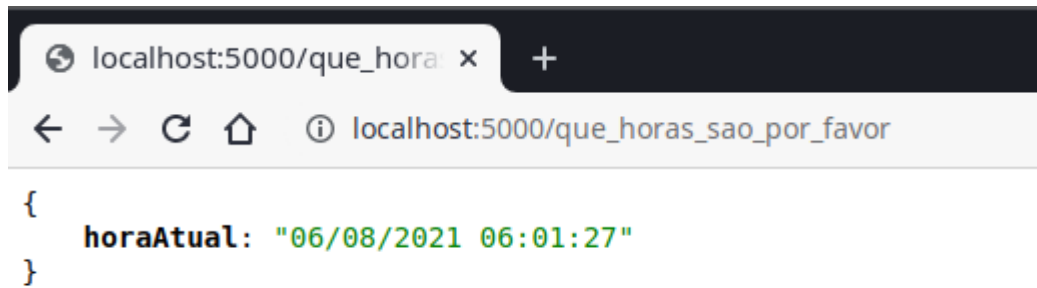


```
[isaac@gru node_only_api]$ node server.js
server iniciou na porta: 5000
```

Para acessar a API você pode usar o navegador:



```
{
  message: "Oi coleguinhas, tudo tranquilo com vocês?"
}
```



Exemplo com o Express

Agora vamos ver a mesma aplicação escrita com o uso do Express.

Para isso você precisa criar uma pasta para a aplicação, entrar na pasta e instalar o express com o npm.

```
[isaac@gru apps]$ mkdir express_simple_api
[isaac@gru apps]$ cd express_simple_api/
[isaac@gru express_simple_api]$ npm install express

added 50 packages, and audited 51 packages in 3s

found 0 vulnerabilities
[isaac@gru express_simple_api]$ |
```

Depois de instalado o express na pasta da sua aplicação você pode criar o arquivo server.js dentro dela com seguinte código:

```
const express = require('express');

const PORT = process.env.PORT || 3000;
const app = express();

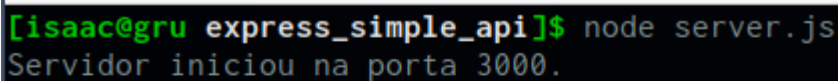
app.get('/ola_servidor', (req, res) => {
  res.json({
    message: 'Oi colegas. tudo tranquilo com vocês?'
  });
});

app.get('/que_horas_sao_por_favor', (req, res) => {
  const dataHora = new Date().toLocaleString();
  res.json({
    horaAtual: dataHora
  });
});
```

```
});  
  
app.get('*', (req, res) => {  
  res.status(404).json({  
    message: 'Rota não encontrada'  
  });  
});  
  
app.listen(PORT, () => {  
  console.log("Servidor iniciou na porta 3000.")  
});
```

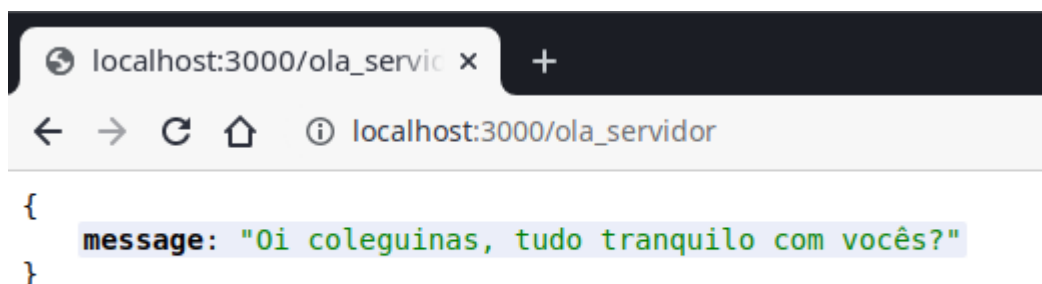
Repare que o código acima usando o express é mais simples pois a verificação das rotas GET são mais fáceis que sem utilizar o Express, a geração do JSON de retorno é mais simples e o código final fica menor e de melhor manutenção. O método "get" do objeto "app" que representa um aplicativo Express, é responsável por definir uma regra para uma rota com o método GET com o primeiro parâmetro sendo a URL da rota e segundo parâmetro uma função que recebe (req, res) que são objetos que definem a requisição e resposta, respectivamente.

Para executar a aplicação:

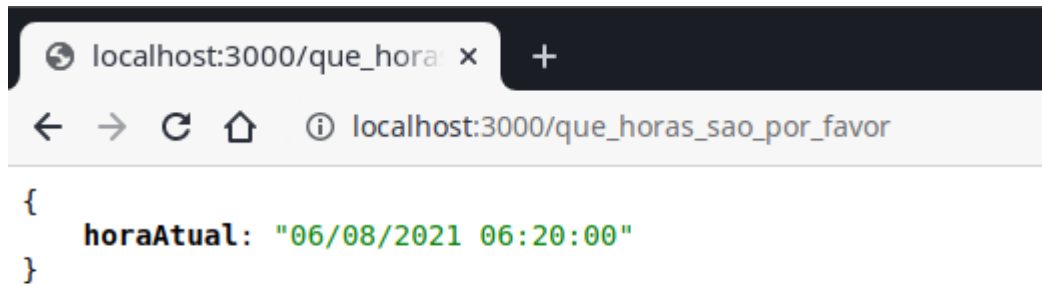


```
[isaac@gru express_simple_api]$ node server.js  
Servidor iniciou na porta 3000.
```

Pode ser acessada também pelo navegador:



```
{  
  message: "Oi colegas, tudo tranquilo com vocês?"  
}
```

Características do Express

Para instalar o Express, vá ao site <https://expressjs.com/> e no menu "Getting starter" você verá instruções detalhadas de instalação e uso do Express.

Vamos ver algumas de suas características.

Um aplicativo Express simples:

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

Na linha `const express = require('express')` o Express é incluído na aplicação.

Na linha `const app = express()` é criado um objeto chamado app que é uma instância do Express.

`const port = 3000` define uma variável que será a porta que o aplicativo irá executar.

O código:

```
app.get('/', (req, res) => {
  res.send('Hello World!')
})
```

Usa o método `get` do objeto `app` para se criar uma rota que responderá na URL `/` rodando uma função definida pelo programador que usa o `res.send` para retornar uma string como resposta para o cliente (navegador ou outra aplicação consumindo uma API, por exemplo).

Por final o código:

```
app.listen(port, () => {  
  console.log(`Example app listening at http://localhost:${port}`)  
})
```

Informa para a aplicação executar na porta definida na variável `port` e assim que ela executar uma mensagem no terminal será exibida.

Express generator

É possível utilizar uma ferramenta chamada `express-generator` para criar rapidamente o esqueleto de uma aplicação Express. Para usar o `express-generator` para criar uma aplicação execute o comando abaixo:

```
npx express-generator --view=ejs MinhaAplicacaoExpress
```

Resultado do comando:

```
[isaac@gru apps]$ npx express-generator --view=ejs MinhaAplicacaoExpress

create : MinhaAplicacaoExpress/
create : MinhaAplicacaoExpress/public/
create : MinhaAplicacaoExpress/public/javascripts/
create : MinhaAplicacaoExpress/public/images/
create : MinhaAplicacaoExpress/public/stylesheets/
create : MinhaAplicacaoExpress/public/stylesheets/style.css
create : MinhaAplicacaoExpress/routes/
create : MinhaAplicacaoExpress/routes/index.js
create : MinhaAplicacaoExpress/routes/users.js
create : MinhaAplicacaoExpress/views/
create : MinhaAplicacaoExpress/views/error.ejs
create : MinhaAplicacaoExpress/views/index.ejs
create : MinhaAplicacaoExpress/app.js
create : MinhaAplicacaoExpress/package.json
create : MinhaAplicacaoExpress/bin/
create : MinhaAplicacaoExpress/bin/www

change directory:
$ cd MinhaAplicacaoExpress

install dependencies:
$ npm install

run the app:
$ DEBUG=minhaaplicacaoexpress:* npm start
```

Repare que foi utilizado o parâmetro "--view=ejs" no comando para informar ao express-generator que as views da aplicação serão com o motor de templates EJS. É possível escolher outros motores de templates como: Pug, Jade, hbs, dust, twig, vash, etc. Essas motores de templates são bibliotecas que facilitam a criação de HTML dinâmico, e cada um tem suas características distintas. Usamos o EJS aqui por opção.

Foi criada uma pasta chamada "MinhaAplicacaoExpress" com diversos arquivos de uma aplicação express.

Foi criada uma pasta "public" para se adicionar os arquivos estáticos (imagens, estilos CSS, etc).

Foi criada uma pasta "routes" com arquivos que representam as rotas da aplicação, definidas em Javascript.

Foi criada também uma pasta chamada "views" com os arquivos EJS que são as páginas dinâmicas que podem ser renderizadas pelas rotas com um comando específico.

Repare que mesmo sendo considerado um framework back-end o Express se

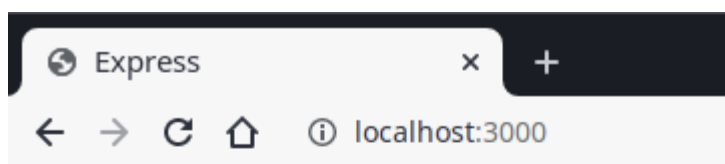
integra bem com motores de templates para a criação de páginas dinâmicas (views), entretanto se deseja acessar bancos de dados, por exemplo, você precisará escolher suas bibliotecas favoritas, o que também é simples com o Express, mas ele não trás nenhuma escolha própria para isso.

Uma aplicação criada com o express-generator já vem com outras adições como o cookie-parser para se manipular cookies, o morgan para se gerar logs de acesso do sistema, o view engine (motor de templates) EJS incluso, os middlewares para se aceitar JSON nas requisições, um provedor de arquivos estáticos configurado na pasta "public" um handler de erros que permite você retornar mensagens personalizadas para cada erro da sua aplicação, etc. Tudo isso pode ser visto no arquivo app.js criado para você.

Para instalar as dependências e rodar a aplicação:

```
[isaac@gru apps]$ cd MinhaAplicacaoExpress/  
[isaac@gru MinhaAplicacaoExpress]$ npm install  
  
added 54 packages, and audited 55 packages in 1s  
  
found 0 vulnerabilities  
[isaac@gru MinhaAplicacaoExpress]$ npm start  
  
> minhaaplicacaoexpress@0.0.0 start  
> node ./bin/www
```

Vendo o resultado no navegador:



Roteamento

As rotas são definidas no express através de métodos que têm o mesmo nome dos métodos HTTP. Veja alguns exemplos da documentação do Express:

Respond with Hello World! on the homepage:

Responde com "Hello World!" em GET "/"

```
app.get('/', function (req, res) {  
  res.send('Hello World!')  
})
```

Responde a um POST em "/"

```
app.post('/', function (req, res) {  
  res.send('Got a POST request')  
})
```

Responde a uma requisição PUT para a rota /user:

```
app.put('/user', function (req, res) {  
  res.send('Got a PUT request at /user')  
})
```

Responde a um DELETE na rota /user:

```
app.delete('/user', function (req, res) {  
  res.send('Got a DELETE request at /user')  
})
```

Para mais informações sobre rotas avançadas acesse:

<https://expressjs.com/en/guide/routing.html>

Middlewares

O Express usa fortemente o conceito de middlewares.

Middlewares são nada mais que funções que executam antes de requisições, caso você deseje.

Por exemplo, você pode desejar que antes de todas as rotas dentro de uma área administrativa do seu sistema, seja feita uma verificação se o usuário está logado e se ele tem permissão de acesso. Isso pode ser feito se criando um middleware que pode ser usado em todas as rotas que precisam e você não precisa programar essa checagem dentro de cada uma delas.

Para uma lista extensa de exemplos de aplicações de exemplo escritas com o

Express visite: <https://expressjs.com/en/starter/examples.html>

```
const express = require('express');

const PORT = process.env.PORT || 3000;
const app = express();

app.get('/hora_atual', (req, res) => {
  res.json({ horaAtual: new Date().toLocaleString() });
});

function ProtegeRota(req, res, next) {
  // Aqui pode ser verificado se o usuário está logado ou qualquer
  // outra restrição.
  // Se tudo estiver ok, basta chamar next() para seguir com a
  // requisição.
  // Caso as restrições não sejam obedecidas você pode finalizar a
  // requisição com res.end();

  if (req.query.codigo_de_acesso == 'CodigoSecreto') {
    next();
  } else {
    return res.send('Não tem acesso').end();
  }
}

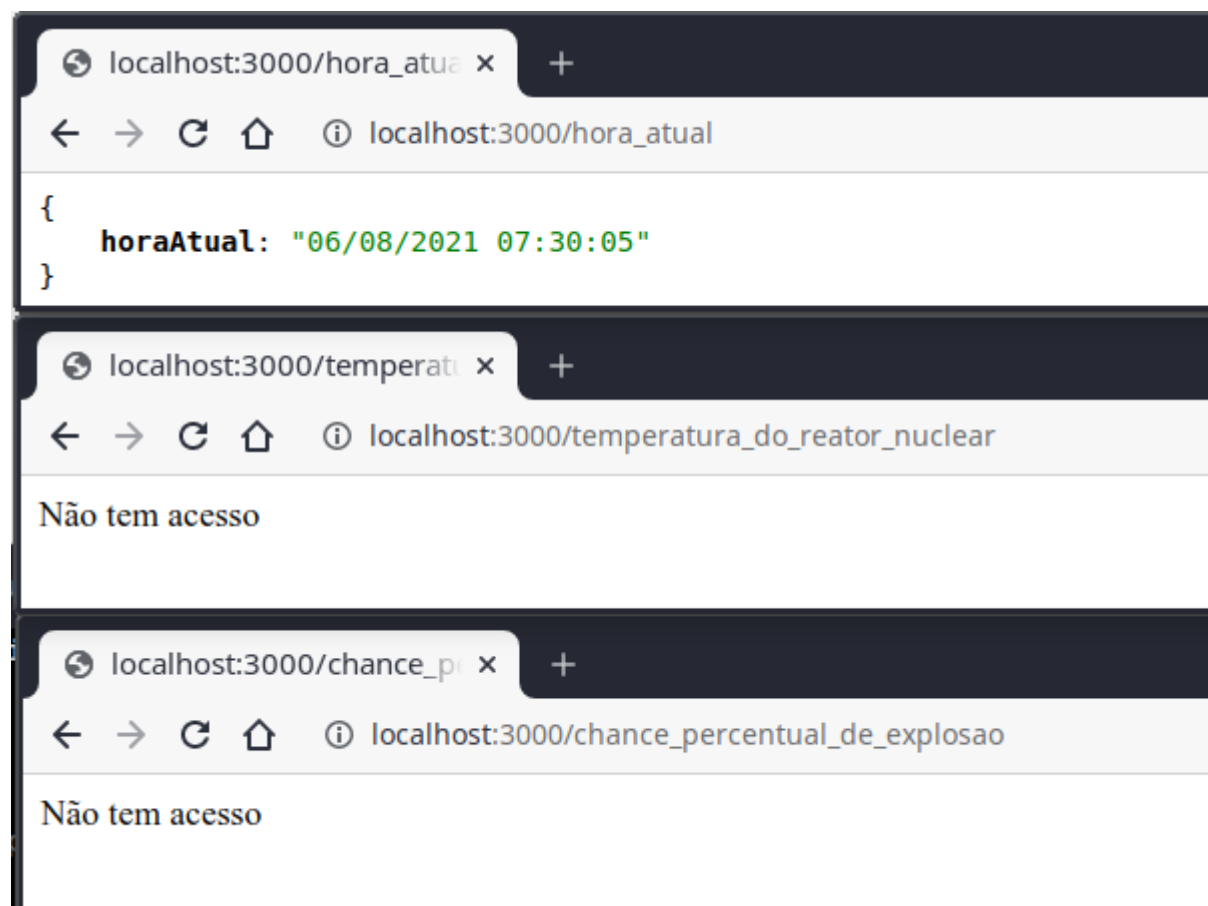
app.get('/temperatura_do_reator_nuclear', [ProtegeRota], (req, res)
=> {
  res.json({ temperatura: 220 });
});

app.get('/chance_percentual_de_explosao', [ProtegeRota], (req, res)
=> {
  res.json({ chance_de_explosao: 95 });
});

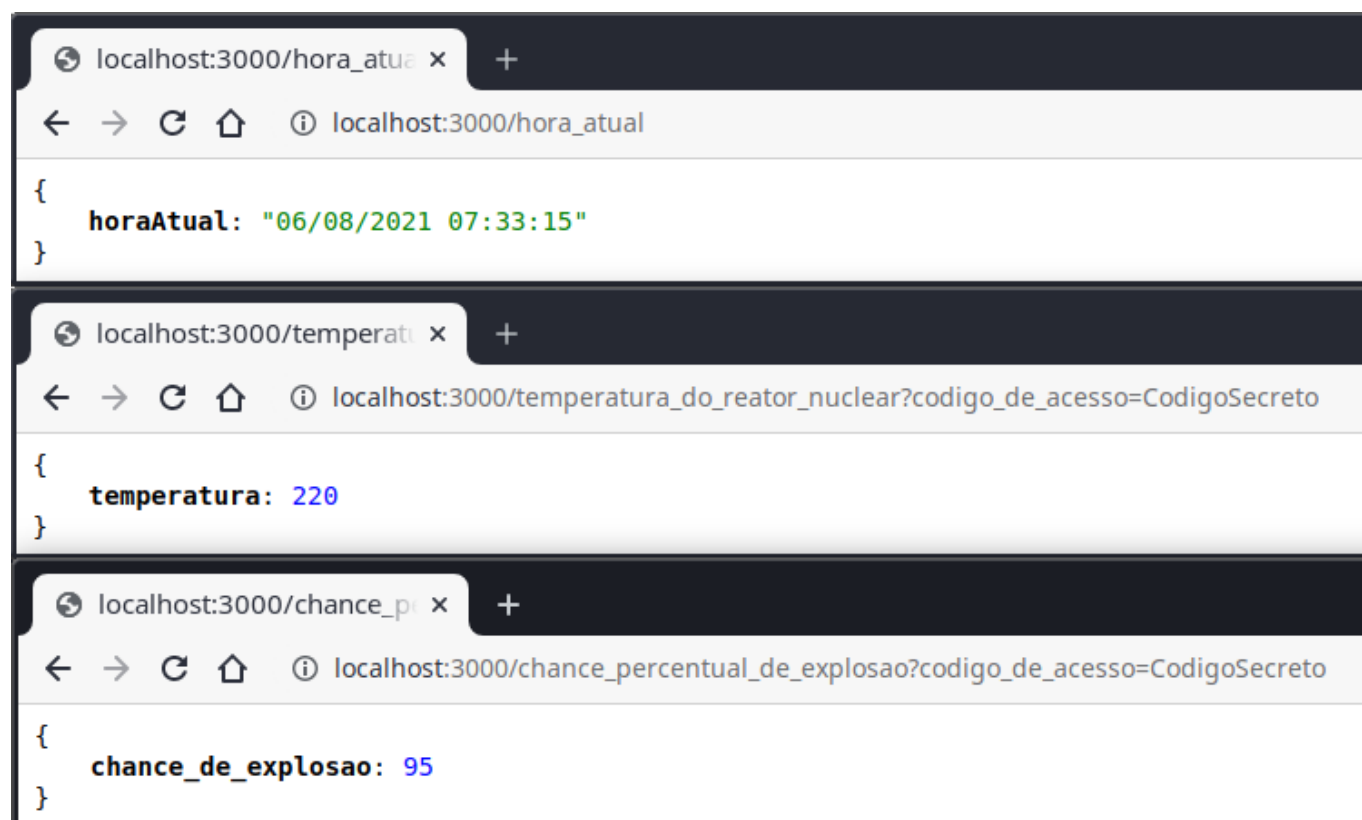
app.listen(PORT, () => {
  console.log("Servidor iniciou na porta 3000.")
});
```

Repare no código acima que criamos um middleware chamado ProtegeRota que foi usado em duas das três rotas do sistema e ele verifica se o usuário passou um código de acesso para permitir que a execução das rotas ocorram.

Acessando:



Acessando com códigos de acesso:



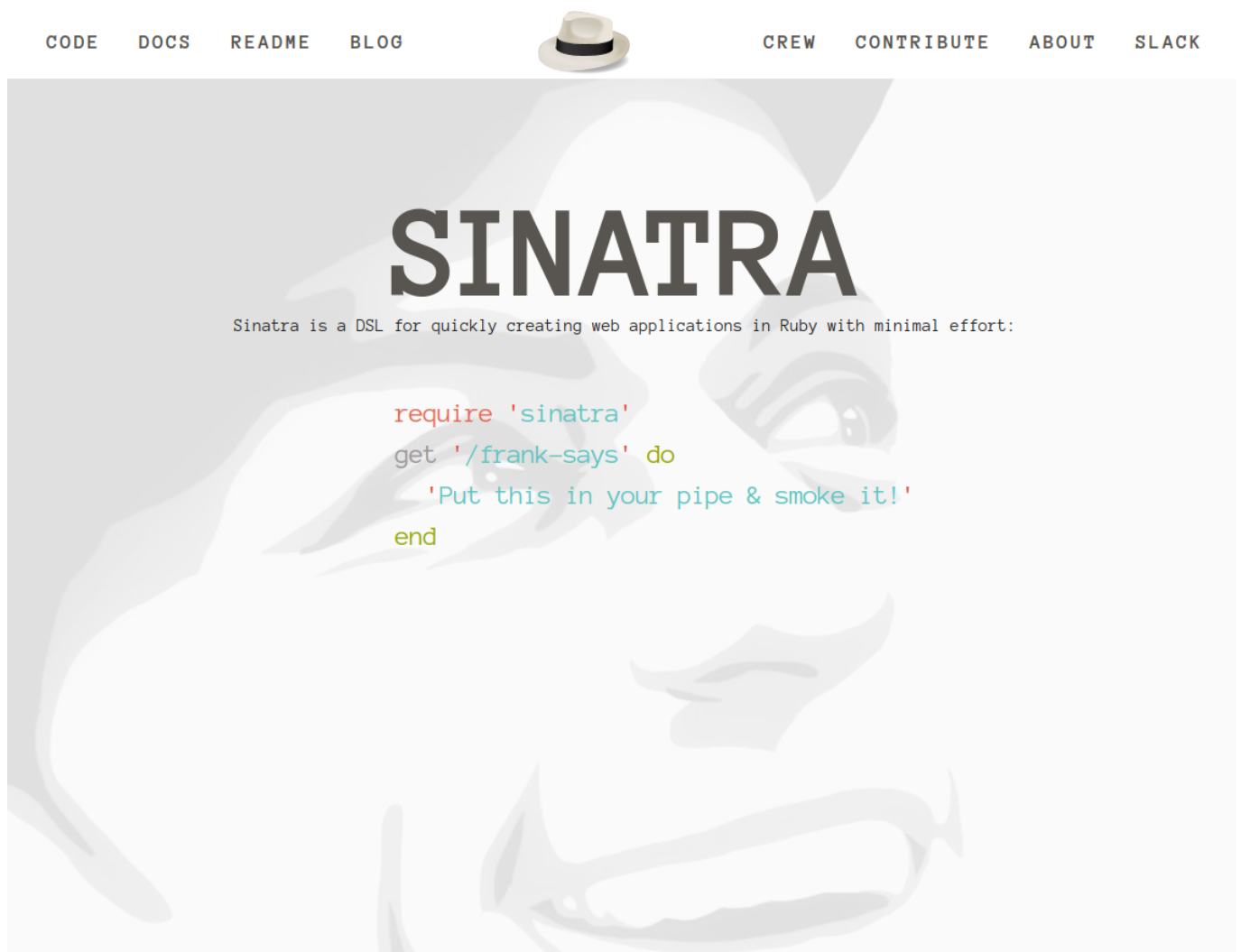
O Express é um framework enxuto, rápido e consegue se integrar bem com

diversas bibliotecas escritas especificamente para ele ou não, aumentando as funcionalidades da ferramenta e a tornando uma excelente alternativa para a construção inclusive de sistemas completos em Javascript.

Sinatra

Web site: <http://sinatrarb.com/>

O Sinatra é um micro-framework para a linguagem de programação Ruby que tem por objetivo ser o mais simples e mínimo possível na criação de serviços que respondem a requisições HTTP. Ele somente cuida do processamento das rotas e executa funções Ruby para cada uma delas, assim como o Express, mas não tem nenhuma outra funcionalidade, cabendo a você adicionar o que desejar.



Website do Sinatra. Fonte: <http://sinatrarb.com/>

Ele utiliza a linguagem de programação Ruby (<https://www.ruby-lang.org>) que pode ser explorada pelo navegador, se desejar aprender mais: <https://try.ruby-lang.org/>

Uma aplicação mínima no sinatra tem um seguinte formato (em Ruby):


```
require 'sinatra'
require 'json'

set :port, 3000

get '/ola_servidor' do
  content_type :json
  { message: 'Oi colega. Tudo em paz?' }.to_json
end

get '/que_horas_sao_por_favor' do
  content_type :json
  { horaAtual: Time.now.strftime("%d/%m/%Y %H:%M") }.to_json
end
```

Abaixo, dois comandos no terminal. Um para instalar o sinatra no sistema, utilizando o comando para instalar de pacotes do Ruby ("gem install sinatra") e outro para executar o programa escrito acima que está no arquivo server.rb:

```
[isaac@gru sinatra]$ gem install sinatra
Fetching sinatra-2.1.0.gem
Successfully installed sinatra-2.1.0
1 gem installed
[isaac@gru sinatra]$ ruby server.rb
[2021-08-06 07:48:07] INFO WEBrick 1.6.0
[2021-08-06 07:48:07] INFO ruby 2.7.2 (2020-10-01) [x86_64-linux]
== Sinatra (v2.1.0) has taken the stage on 3000 for development with backup from WEBrick
[2021-08-06 07:48:07] INFO WEBrick::HTTPServer#start: pid=26114 port=3000
```

O comando `require 'sinatra'` inclui o sinatra no seu programa Ruby. O comando `require 'json'` adiciona suporte a conversão de objetos para JSON no Ruby com o método `.to_json`.

`set :port, 3000` configura a porta do sinatra para ser a 3000.

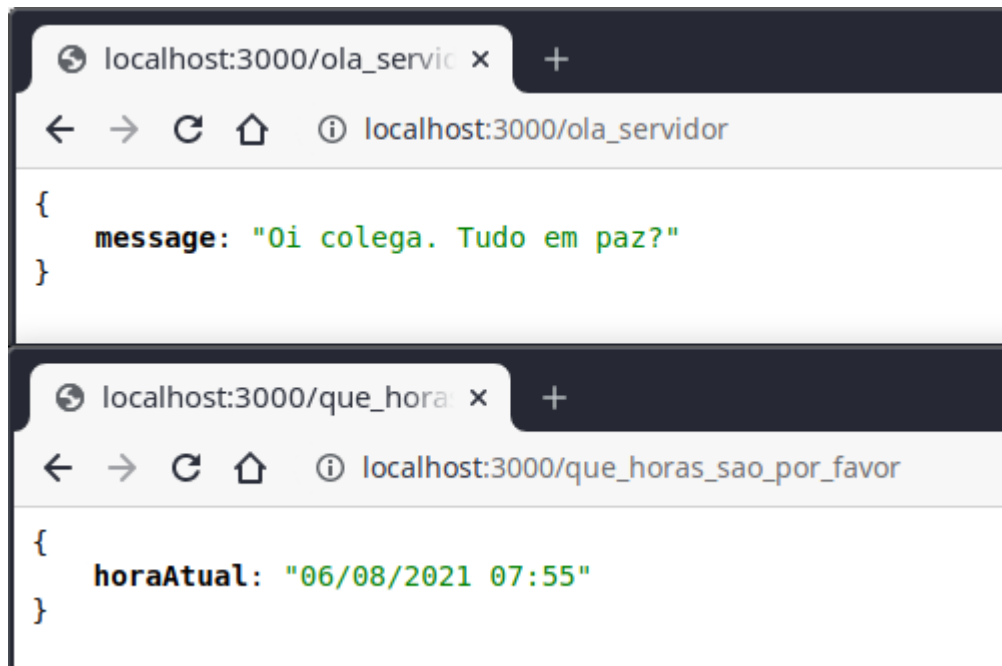
As funções `get` são chamadas com o parâmetro passado como uma string representando a rota que deseja processar. No Ruby não é necessário usar parênteses para se executar funções e os limitadores "do" e "end" definem o início e fim da função, nesse caso é um bloco de código que está sendo executado quando essa rota é acessada.

`content_type :json` adiciona no Header da resposta o tipo de conteúdo como JSON

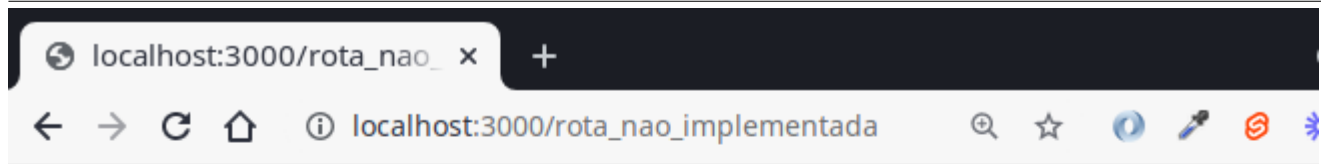
A última linha de cada bloco é o retorno do bloco, no caso da aplicação acima é {

message: 'Oi colega. Tudo em paz?' }.to_json, ou seja, um objeto Ruby convertido para uma string json que vai para o navegador.

Acessando no navegador:



O Sinatra tem um comportamento padrão (que você pode mudar, claro) quando se acessa uma rota inexistente. Vamos tentar:



Sinatra doesn't know this ditty.



Try this:

```
get '/rota_nao_implementada' do  
  "Hello World"  
end
```

Veja que ele mostra uma mensagem de erro estilizada e inclusive sugere que código você deve criar na sua aplicação para que essa rota não implementada seja criada com sucesso.

Características do Sinatra

O Sinatra permite a criação de sistemas web em Ruby com esforço realmente mínimo. Vamos ver algumas funcionalidades.

Rotas

No Sinatra, uma rota é um método HTTP junto com uma padrão de URL a ser comparado. Cada rota é associada com um bloco de código Ruby. Veja alguns exemplos:

```
get '/' do
  'mostra alguma coisa'
end

post '/' do
  'cria algum dado'
end

put '/' do
  'exemplo de um put'
end

patch '/' do
  'exemplo de um patch'
end

delete '/' do
  'exemplo de um delete'
end
```

Rotas podem ter parâmetros, assim como no Express, por exemplo. Veja um exemplo:

```
get '/oi/:nome' do
  "Olá #{params['nome']}!"
end
```

No exemplo acima a rota casa com requisições como `"/oi/joao"` ou `"/oi/maria"`, etc. O que for passado depois de `/oi/` fica armazenado em `params['nome']`. No Ruby para se criar strings com variáveis ou expressões concatenadas no meio a sintaxe é essa vista no código: `"Algum texto mostrando uma #{variavel_ou_expressao}"`

É possível criar regras avançadas nos padrões dinâmicos das rotas com o Sinatra. Para mais informações acesse: <http://sinatrarb.com/intro.html>

Criando uma API com o Sinatra

Vamos rapidamente mostrar os comandos para se criar um banco de dados chamado `loja_de_animais` no MariaDB (com o conector do MySQL) com uma tabela chamada `animais`. Em seguida vamos criar uma aplicação sinatra com uma API para criar animais e listar os animais criados.

Criando o banco de dados e tabela (Requer o MariaDB ou MySQL instalado)

```
[isaac@gru sinatra]$ mysql -uroot -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 22
Server version: 10.6.3-MariaDB Arch Linux

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> create database loja_de_animais;
Query OK, 1 row affected (0.000 sec)

MariaDB [(none)]> use loja_de_animais;
Database changed
MariaDB [loja_de_animais]> create table animais(id integer primary key auto_increment,
nome varchar(255), tipo varchar(255), raca varchar(255), descricao TEXT);
Query OK, 0 rows affected (0.019 sec)

MariaDB [loja_de_animais]> exit
Bye
[isaac@gru sinatra]$ |
```

Os comandos realizados acima, são detalhados abaixo. No primeiro, conecta-se ao banco de dados pelo terminal:

```
# mysql -uroot -p
Enter password: Sua senha aqui
```

O comando `create database loja_de_animais;` cria uma base de dados no MariaDB ou MySQL

```
MariaDB [(none)]> create database loja_de_animais;
Query OK, 1 row affected (0.000 sec)
```

`MariaDB [(none)]> use loja_de_animais;` informa que você deseja usar o banco loja_de_animais.

`create table animais(id integer primary key auto_increment, nome varchar(255), tipo varchar(255), raca varchar(255), descricao TEXT);` cria a tabela de animais no banco, com as colunas id, nome, tipo, raca e descricao.

Pronto, nosso banco está criado. Agora instalar três pacotes (gems) do Ruby para nossa aplicação: Sinatra (caso já não tenha feito), o ActiveRecord (ORM Ruby para acesso a banco de dados) e o mysql2 (driver do mysql/mariadb que o ActiveRecord vai utilizar). Vamos usar o comando `gem install sinatra activerecord mysql2` para instalar todos os pacotes de uma vez. Veja o resultado:

```
# gem install sinatra activerecord mysql2
```

```
Successfully installed sinatra-2.1.0
Successfully installed activerecord-6.1.4
Building native extensions. This could take a while...
Successfully installed mysql2-0.5.3
3 gems installed
```

Já com o Sinatra, ActiveRecord e mysql2 instalado, vamos agora criar a API em um arquivo chamado `animais.rb` com rotas para criar (POST `/animais`) e listar animais (GET `/animais`) e remover um animal por id (DELETE `/animal/:id`)

Já com o Sinatra instalado em uma pasta, vamos criar um arquivo com o nome `animais.rb` com toda a aplicação nele.

```
require 'sinatra'
require 'active_record'

set :port, 3000

# Estabelece conexão com banco de dados com suas credenciais.
# Só executa uma vez quando a aplicação inicia.
ActiveRecord::Base.establish_connection({
  adapter: 'mysql2',
  host: 'localhost',
  username: 'USUARIO DO BANCO',
  password: 'SENHA DO BANCO',
  database: 'loja_de_animais'
})

# Essa classe é um Model do ORM ActiveRecord e
# está configurada para acessar a tabela 'animais'
# Com isso ela consegue manipular essa tabela completamente.
class Animal < ActiveRecord::Base
  self.table_name = 'animais'
end

get '/animais' do
  animais = Animal.all # Utilizando a classe Animal obtém todos os
    cadastrados no banco
  return animais.to_json # retorn um JSON com todos os animais da
    base de dados
end

post '/animais' do
  # recebe os dados JSON da requisição convertendo para um objeto
```

Ruby

```
dados = JSON.parse(request.body.read)

animal = Animal.new # Cria uma instância da classe Animal

# setar o nome, tipo, raca e descricao para os passado no POST
animal.nome = dados['nome']
animal.tipo = dados['tipo']
animal.raca = dados['raca']
animal.descricao = dados['descricao']

# salva no banco de dados
animal.save()
return { message: 'Animal criado com sucesso!', animal: animal
}.to_json # Retorna um JSON com os dados do animal salvo
end

# Remove um animal por ID
delete '/animal/:id' do
  #Busca um animal com ID passado na rota como parâmetro do método
  find
  animal = Animal.find(params['id'])
  animal.destroy() #remove o animal da base de dados
  return { message: "#{animal.nome} foi removido com sucesso"
}.to_json
end
```

A API está criada. Vamos utilizar o Insomnia (pode ser baixado em <https://insomnia.rest/>) para a acessar e realizar várias operações:

POST <http://localhost:3000/animais> Send

JSON Auth Query Header 1 Docs

```
1 {  
2   "nome": "Bruce Wayne",  
3   "tipo": "Cão",  
4   "raca": "Pug",  
5   "descricao": "Cão estranho e fofo ao mesmo tempo"  
6 }
```

Beautify JSON

200 OK 59.9 ms 156 B 2 Minutes Ago

Preview Header 8 Cookie Timeline

```
1 {  
2   "message": "Animal criado com sucesso!",  
3   "animal": {  
4     "id": 4,  
5     "nome": "Bruce Wayne",  
6     "tipo": "Cão",  
7     "raca": "Pug",  
8     "descricao": "Cão estranho e fofo ao mesmo tempo"  
9   }  
10 }
```

Cadastrando Bruce Wayne

(POST <http://localhost:3000/animais>)

POST <http://localhost:3000/animais> Send

JSON ▾ Auth ▾ Query Header 1 Docs

```
1 {
2   "nome": "Baleia",
3   "tipo": "Cão",
4   "raca": "Mistério",
5   "descricao": "Um animal cor de café com leite de origens
6   desconhecidas mas com muito amor para oferecer!"
7 }
```

Beautify JSON

200 OK 6.34 ms 213 B Just Now ▾

Preview ▾ Header 8 Cookie Timeline

```
1 {
2   "message": "Animal criado com sucesso!",
3   "animal": {
4     "id": 5,
5     "nome": "Baleia",
6     "tipo": "Cão",
7     "raca": "Mistério",
8     "descricao": "Um animal cor de café com leite de origens
9     desconhecidas mas com muito amor para oferecer!"
10  }
```

Cadastrando Baleia (POST

<http://localhost:3000/animais>)

POST <http://localhost:3000/animais> Send

JSON ▾ Auth ▾ Query Header 1 Docs

```
1 {
2   "nome": "Oriosvaldo",
3   "tipo": "Ornitorrinco",
4   "raca": "Padrão",
5   "descricao": "Um bixo com bico de pato, esporão de galo,
6   nada, e tem bolsa de canguro... querendo comprar, temos."
7 }
```

Beautify JSON

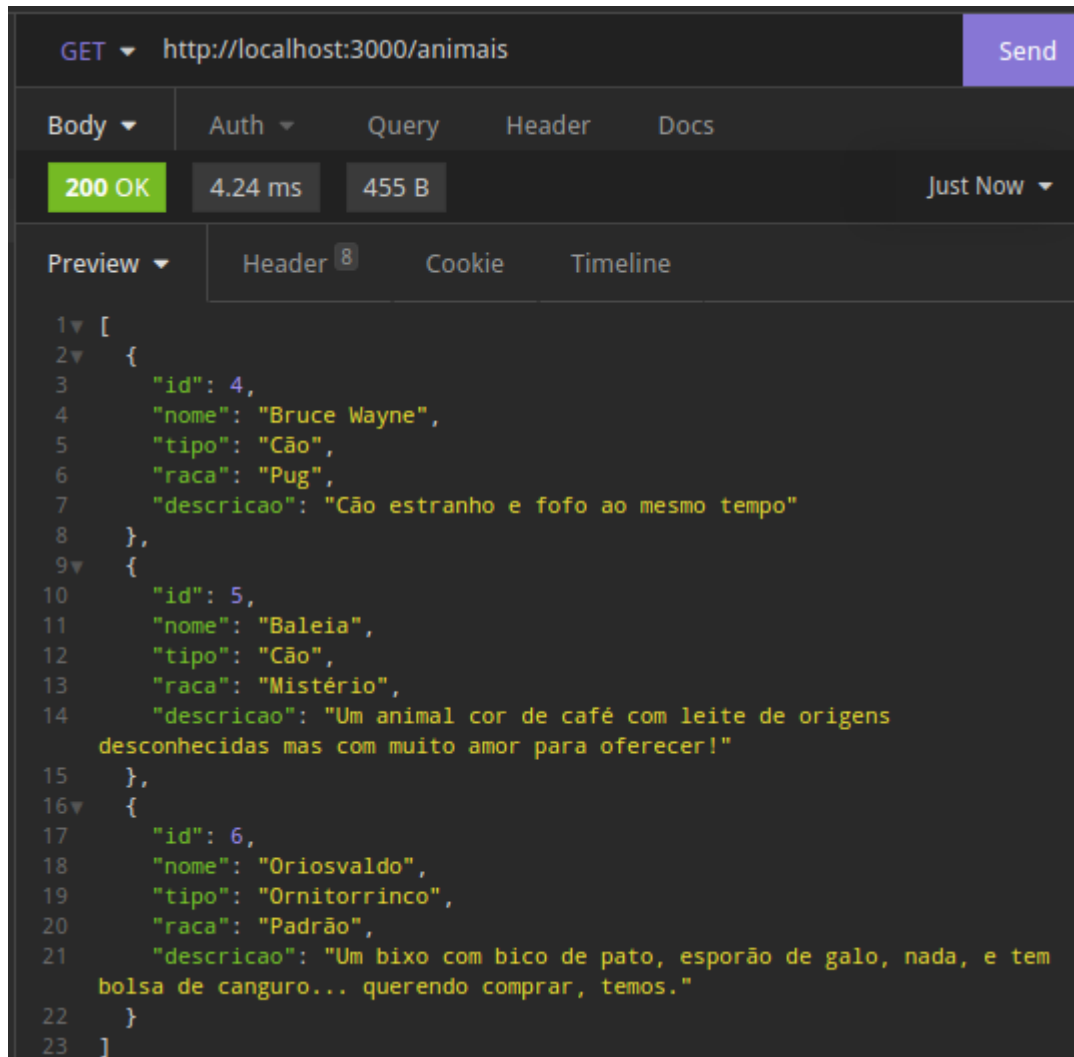
200 OK 8.42 ms 232 B Just Now ▾

Preview ▾ Header 8 Cookie Timeline

```
1 {
2   "message": "Animal criado com sucesso!",
3   "animal": {
4     "id": 6,
5     "nome": "Oriosvaldo",
6     "tipo": "Ornitorrinco",
7     "raca": "Padrão",
8     "descricao": "Um bixo com bico de pato, esporão de galo,
9     nada, e tem bolsa de canguro... querendo comprar, temos."
10  }
```

Cadastrando Oriosvaldo

(POST <http://localhost:3000/animais>)



```
GET http://localhost:3000/animais Send

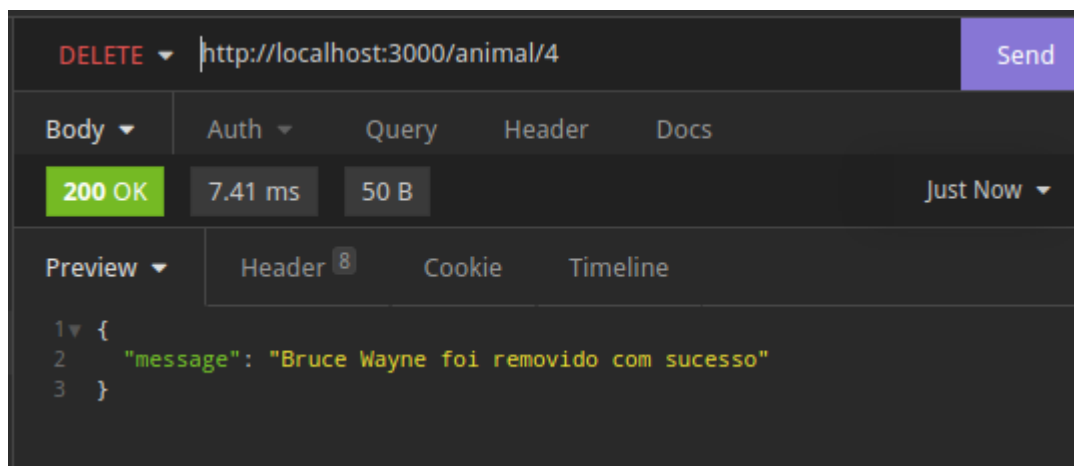
Body Auth Query Header Docs

200 OK 4.24 ms 455 B Just Now

Preview Header 8 Cookie Timeline

1 [
2   {
3     "id": 4,
4     "nome": "Bruce Wayne",
5     "tipo": "Cão",
6     "raca": "Pug",
7     "descricao": "Cão estranho e fofo ao mesmo tempo"
8   },
9   {
10    "id": 5,
11    "nome": "Baleia",
12    "tipo": "Cão",
13    "raca": "Mistério",
14    "descricao": "Um animal cor de café com leite de origens desconhecidas mas com muito amor para oferecer!"
15  },
16  {
17    "id": 6,
18    "nome": "Oriosvaldo",
19    "tipo": "Ornitorrinco",
20    "raca": "Padrão",
21    "descricao": "Um bixo com bico de pato, esporão de galo, nada, e tem bolsa de canguro... querendo comprar, temos."
22  }
23 ]
```

Obtendo a lista de animais cadastrados: (GET <http://localhost:3000/animais>)



```
DELETE http://localhost:3000/animal/4 Send

Body Auth Query Header Docs

200 OK 7.41 ms 50 B Just Now

Preview Header 8 Cookie Timeline

1 {
2   "message": "Bruce Wayne foi removido com sucesso"
3 }
```

Removendo um animal por ID: DELETE <http://localhost:3000/animal/4>

Pronto. Criamos um API do início ao fim, desde o banco de dados, tabela, aplicação com o Sinatra e consumimos os dados com o Insomnia realizando as operações de criar animal, listar e remover um animal por ID.

Como você viu utilizamos a biblioteca ActiveRecord dentro do Sinatra para acessar o banco de dados. O ActiveRecord é uma biblioteca originalmente criada para o

framework RubyOnRails, mas como é modularizada ela pode ser utilizada fora do Rails, como fizemos aqui. Mas isso não é realmente necessário. Seria possível usar o driver do mysql diretamente no nosso código. Usamos o ActiveRecord somente para ter mais facilidades.

Conclusão

Nessa aula tivemos uma visão geral do que são frameworks back-end. São muitas opções, em muitas linguagens diferentes. O importante é notar que é possível sim iniciar sistemas web com configurações menores, com menos código e mais simples, e ao longo do tempo ir adicionando mais bibliotecas com mais funcionalidades prontas.

Exploramos com maior detalhes o Express e o Sinatra em exemplos e se você desejar pode criar as mesmas funcionalidades no seu computador.