
# REST

# REST - o que é?

REST é um modelo para ser utilizado ao projetar arquiteturas de software distribuído.

Não é uma biblioteca ou tecnologia, se utiliza de chamadas HTTP.

Foi criada por Roy Fielding em 2000 na sua dissertação de doutorado.

# Baseado em recursos

REST é baseado em recursos, não em ações.

Exemplo de recursos: Pessoa, Carro.

Exemplo de ações: buscaDadosPessoa, adicionaCarro

# Representação de recursos

Os dados transmitidos entre o cliente e servidor são a representação de um recurso.

Esta representação pode ser o recurso completo ou parte dele. Os formatos mais comuns são JSON e XML.

Ex: GET /carros/1 -> { placa: "FGG-3433", marca: "Mercedes"}

O Json retornado para o cliente é a representação do recurso carro.

# RESTful API

Uma API apenas pode ser considerada RESTful se atender as seguintes constraints:

- Interface uniforme
- Separação entre cliente e servidor
- Sem estado
- Cacheable
- Dividido em Camadas

AvenueCode

# Interface uniforme

Uma API rest deve ter uma interface uniforme para acesso aos recursos, contendo os seguintes itens:

Verbo HTTP: GET, POST, PUT, DELETE ...

URI (identificador do recurso: Ex: /carros, /pessoas

Resposta HTTP: Status code e corpo. Ex: Status code 200 (sucesso)

# Separação entre cliente e servidor

O cliente e o servidor REST agem independentemente e a única interação entre eles é através de requests feitos pelo client.

# Sem estado

Uma API rest não deve guardar dados de sessão de usuário.

Toda a chamada para uma API rest contém tudo o que é necessário para o processamento.

# Cacheable

Muitas respostas de uma API Rest podem permanecer as mesmas para várias requisições.

É responsabilidade do servidor informar se a representação de um recurso pode ser guardada em cache e qual o período de expiração.

# Dividido em camadas

Entre o cliente e servidor, podem existir várias camadas (balanceamento de carga, segurança, cache, etc).

O cliente não se importa ou conhece essas camadas, ele simplesmente faz a chamada e recebe a resposta.

# Componentes de uma request

Em uma request haverá os seguintes componentes:

- URI - Caminho que representa o recurso
- RequestBody - Sua requisição pode ter um corpo em um formato específico
- Metadados:
  a) Verbo: GET, POST etc…
  b) Content-type: Tipo do RequestBody: application/json, application/xml, etc…
  c) Accept: Content-type desejado no retorno.

# Verbos HTTP

- GET: Utilizado para retornar uma lista de recursos ou um recurso individual.
- POST: Utilizado para criar um novo recurso.
- PATCH: Utilizado para modificar um recurso parcialmente.
- PUT: Utilizado para atualizar um recurso existente.
- DELETE: Utilizado para excluir um recurso.

# PathParam e QueryParam

QueryParams e PathParams são utilizados para enviar parâmetros via URL no request.

# QueryParam

QueryParams são enviados através de chave-valor após o símbolo ? no final da URL.

Exemplo: /carros?modelo=chevrolet -> envia o parâmetro modelo igual a chevrolet.

# PathParam

PathParams são enviados no meio da URL para identificar recursos.

Exemplo: /carros/12 -> requisita o recurso com id 12

# Componentes de uma response

1) Metadados de resposta:
   a) Content-type: Formato da resposta do servidor
   b) HTTP Status: Código de resposta do servidor, que identifica o status de sucesso ou falha da requisição.

2) ResponseBody: A resposta pode ter um corpo em um formato específico

AvenueCode

# Códigos de resposta

2xx: Indica que o request foi aceito e executado com sucesso.

3xx: Indica que o client deve tomar ações extras para completar o request.

4xx: Ocorreu um erro em que a responsabilidade é do cliente.

5xx: Ocorreu um erro em que a responsabilidade é do servidor.

# Códigos de resposta mais utilizados

- 200 (OK) - Código geral para indicar sucesso no request.
- 201 (Created) - Indica que um recurso foi criado.
- 204 (No content) - Indica sucesso e que não há nenhum dado no response body.
- 400 (Bad request) - Código geral quando há algum problema com o request do client.
- 401 (Unauthorized) - Indica que falta dados de autenticação no request.

# Códigos de resposta mais utilizados

- 403 (Forbidden) - Indica que o cliente não tem permissão de acessar o recurso.
- 404 (Not found) - Indica que o recurso não existe.
- 405 (Method not allowed) - Indica que a URL existe, mas a API não permite o método HTTP do request.
- 500 (Internal server error) - Código genérico para indicar erro no servidor ao processar o request.
- 503 (Service unavailable) - Indica que uma API externa não está disponível.

# Headers

Entre os metadados (tanto no envio, quanto na resposta) ainda haverá Headers que poderão ser utilizados para fornecer mais informações sobre a requisição, como dados de autenticação, ou qualquer outra informação desejada.

# Idempotência

Uma operação idempotente significa que ela retorna sempre o mesmo resultado, independente do número de vezes que é chamada.

int a;

a = 4; //operacao idempotente

a++; // operação não idempotente

# Métodos seguros

Métodos seguros são métodos que não modificam nenhum recurso no servidor.

# Idempotência e Métodos seguros

| Método | Idempotente | Seguro |
|---|---|---|
| GET | | |
| POST | NÃO | NÃO |
| PUT | | NÃO |
| DELETE | | NÃO |
| PATCH | NÃO | NÃO |

AvenueCode

# Boas práticas

- Utilizar plural para identificação de recursos. Exemplo: /carros.
- Prover filtro, ordenação, paginação e seleção de campos utilizando query params.
  Exemplos:
  GET /carros?cor=vermelha - retorna carros da cor vermelha
  GET /carros?sort=modelo - retorna carros ordenados por modelo
  GET /carros?page=1&size=10 - retorna a primeira página de carros com tamanho 10
  GET /carros?fields=placa,marca,modelo - retorna a placa, marca e modelo de todos os carros

# Exemplo

API para gerenciar uma revendedora de carros usados.

Operações necessárias:

- Listar carros
- Criar um carro
- Atualizar um carro
- Excluir um carro

# Listar carros

- GET /carros
- Retorna uma lista de carros no formato json.

# Criar um carro

- POST /carros
- Request body deve conter a representação de um recurso carro com todos os dados obrigatórios.

  Ex: {"placa": "FFG-2323", "marca": "Chevrolet", "modelo": "Corsa"}

- Retorna uma resposta vazia com os seguintes códigos de resposta:
- 201: Carro criado com sucesso. Header Location deve apontar para o recurso criado.
- 400: Representação do recurso inválida.
- 500: Erro inesperado no servidor ao criar um carro.

# Atualizar um carro

- PUT /carros/{id}
- Request body deve conter a representação de um recurso carro com todos os dados obrigatórios.

  Ex: {"marca": "Chevrolet", "modelo": "Corsa"}

- Retorna uma resposta vazia com os seguintes códigos de resposta:
- 204: Carro atualizado com sucesso
- 400: Representação do recurso inválida.
- 500: Erro inesperado no servidor ao criar um carro.

# Excluir um carro

- DELETE /carros/{id}
- Retorna uma resposta vazia com os seguintes códigos de resposta:
- 204: Carro excluido com sucesso.
- 500: Erro inesperado no servidor ao excluir um carro.

# Exercício

Desenvolver o controller para uma API que tem operações sobre o recurso carro.

Utilizar o projeto: https://gitlab.com/henriquels25/revendedora/

A camada de serviço já está desenvolvida. Deve ser criado um controller no pacote "controller" e injetado nele a classe "CarroService".

# Exercício

Ao responder o POST, a API deve retornar o código 201 com o header de location do recurso criado.

Ao responder o PUT, a API deve retornar o código 204.

Nos métodos POST e PUT, caso o cliente não enviar placa, ID do modelo ou o modelo não existir, o sistema deve retornar o código 400.

Ao responder o DELETE, a API deve retornar o código 204. Se o veículo não existir na base de dados, a API  deve retornar o código 204.

# Exercício

O projeto contém uma suíte com testes end-to-end que faz chamadas para API e valida diversos cenários.

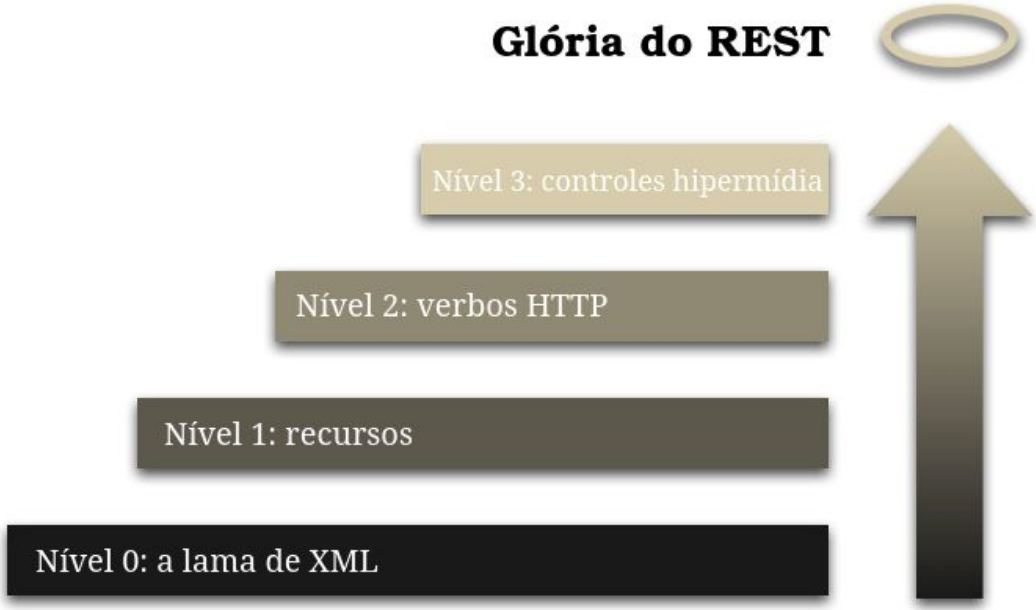
Ao concluir o exercício, fazer o push de uma branch com seu nome para o Gitlab. Será executada uma pipeline com todos os testes.

Para executar os testes end-to-end, pode-se utilizar o seguinte comando:

./mvnw test

AvenueCode

# Modelo de maturidade de Richardson




AvenueCode

# HATEOAS (Hypermedia as the Engine of Application State)

API que guia o cliente durante o seu uso. Cada recurso possui links para os endpoints relacionados.

# Exemplo de HATEOAS


```
{
    "content": [ {
        "price": 499.00,
        "description": "Apple tablet device",
        "name": "iPad",
        "links": [ {
            "rel": "self",
            "href": "http://localhost:8080/product/1"
        } ],
        "attributes": {
            "connector": "socket"
        }
    }, {
        "price": 49.00,
        "description": "Dock for iPhone/iPad",
        "name": "Dock",
        "links": [ {
            "rel": "self",
            "href": "http://localhost:8080/product/3"
        } ],
        "attributes": {
            "connector": "plug"
        }
    } ],
    "links": [ {
        "rel": "product.search",
        "href": "http://localhost:8080/product/search"
    } ]
}
```


Rel: Descreve como o recurso atual é relacionado com o link.

Href: URL para o recurso relacionado.

# Referências

http://blog.caelum.com.br/rest-principios-e-boas-praticas/
https://www.restapitutorial.com/lessons/whatisrest.html
https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-2-rest-constraints-129a4b69a582
https://medium.freecodecamp.org/restful-services-part-ii-constraints-and-goals-530b8f6298b9

AvenueCode

# Referências


https://www.dineshonjava.com/what-is-rest-and-rest-architecture-and-rest-constraints/

https://martinfowler.com/articles/richardsonMaturityModel.html

https://restfulapi.net/hateoas/


AvenueCode