

4.1. O que é REST

quarta-feira, 8 de fevereiro de 2023

11:48

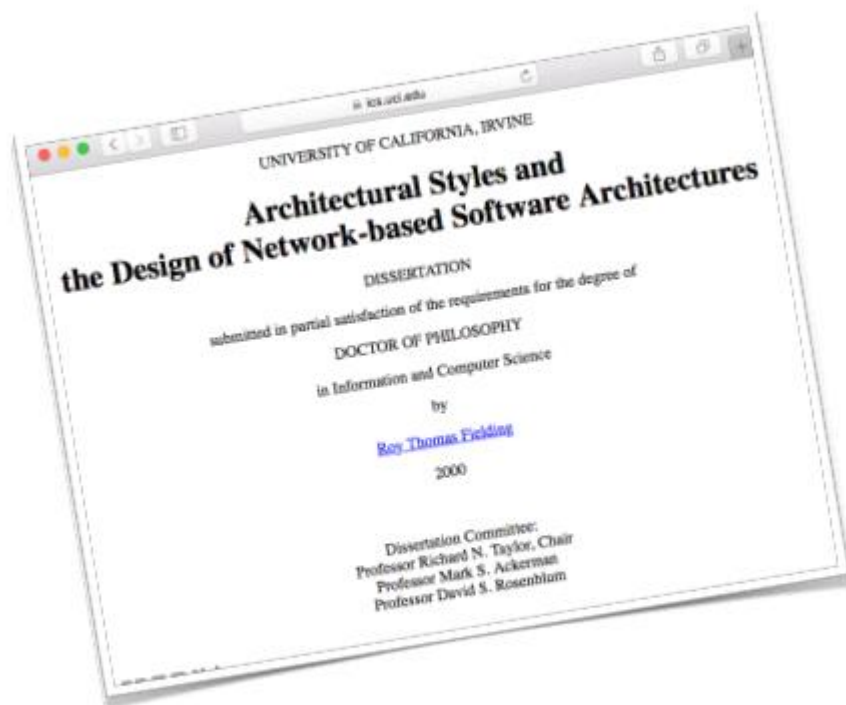


O que é REST?

REST

REpresentational State Transfer

É um modelo arquitetural, é uma especificação que define a forma de comunicação entre componentes de softwares independente da linguagem de programação. Estilo arquitetural para desenvolvimento de web-services.

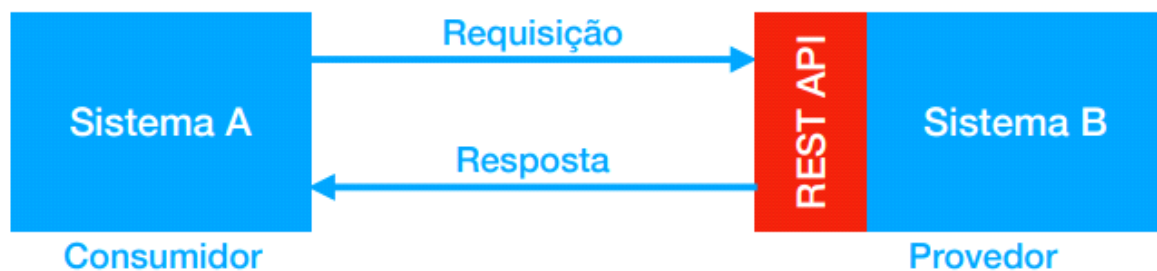


Roy Fielding

Surgiu nos anos 2000 a partir de uma tese de doutorado de Roy Fielding, o intuito da tese era a formalização de conjuntos de **melhores práticas** para desenvolver web-services.

Tais **melhores práticas** são chamadas de **constraints**.

Uma RESTapi é uma api que segue as melhores práticas no modelo arquitetural REST.



Consumidor conhecido como cliente e Provedor conhecido como Servidor.

Por que REST?



- ✓ Separação entre cliente e servidor
- ✓ Escalabilidade
- ✓ Independência de linguagem
- ✓ Mercado

- **SEPARAÇÃO ENTRE CLIENTE E SERVIDOR**

Entre quem consome e quem provê a api, dessa forma temos uma maior flexibilidade e portabilidade, tendo em vista que o sistema provedor e consumidor podem evoluir de formas independentes.

- **ESCALABILIDADE**

Podemos escalar facilmente adicionando outro servidor sem estar replicando sessões.

- **INDEPENDÊNCIA DA LINGUAGEM**

Dentro do mesmo ambiente da empresa podem ser implementados diferentes APIS com linguagens diferentes.

- **MERCADO**

Alta demanda de terceiros para integrar apis.

4.2. Conhecendo as constraints do REST

quarta-feira, 8 de fevereiro de 2023 12:00



Constraints do REST

REST tem o intuito de formalizar um conjunto de constraints, que na verdade são melhores práticas. Elas são:

- **Cliente-servidor**

Precisamos de um cliente para consumir a api e um servidor para provê a api, sendo eles independentes um do outro que e possam evoluir sem alterar o resultado final da aplicação, incluindo, podendo ser substituíveis, desde que, a interface de comunicação deles se mantenham a mesma.

- **Stateless**

Sem-estado - Roy se inspirou no protocolo HTTP para definir essa constraint, na prática quer dizer que a requisição feita ao servidor, deve conter tudo que for necessário para atender completamente a requisição, sem armazenar informações do cliente da requisição, ou seja, o servidor não deve armazenar informações contextuais do cliente que está requisitando a api, ex: Reconhecer o cliente, qual a última requisição que fez, histórico de uso.

- **Cache**

A api pode fazer cache das requisições para diminuir o número de acessos no servidor.

- **Interface uniforme**

Conjunto de operações bem definidas no sistema, uma vez definida como a interface da api funciona, deverá ser seguido sempre, simplificando e desacoplando a arquitetura fazendo com que cada parte evolua de forma independente. E para ficar de acordo, devemos identificar as coisas do sistema usando URI, além de usar o padrão de protocolo para a comunicação como o HTTP (verbos get, post, put, delete) e devemos apontar links para outros recursos (HATEOAS). A resposta da requisição deve ser padronizada especificando formas de como o cliente deve tratar a mensagem, resumindo, a api deve ser feita com os verbos http

corretamente, de forma que a interface funcione como um contrato onde o cliente-servidor se comuniquem de forma mais previsível.

- **Sistema em Camadas**

A constraint de sistema em camadas indica que na comunicação entre cliente servidor, podem haver camadas entre o servidor que envia ou recebe a requisição e o cliente, tais servidores pode fazer qualquer tipo de tratamento como balanceamento de cargas, cache, segurança. Tais camadas não devem interferir na comunicação e o cliente não deve conhecer tais camadas.

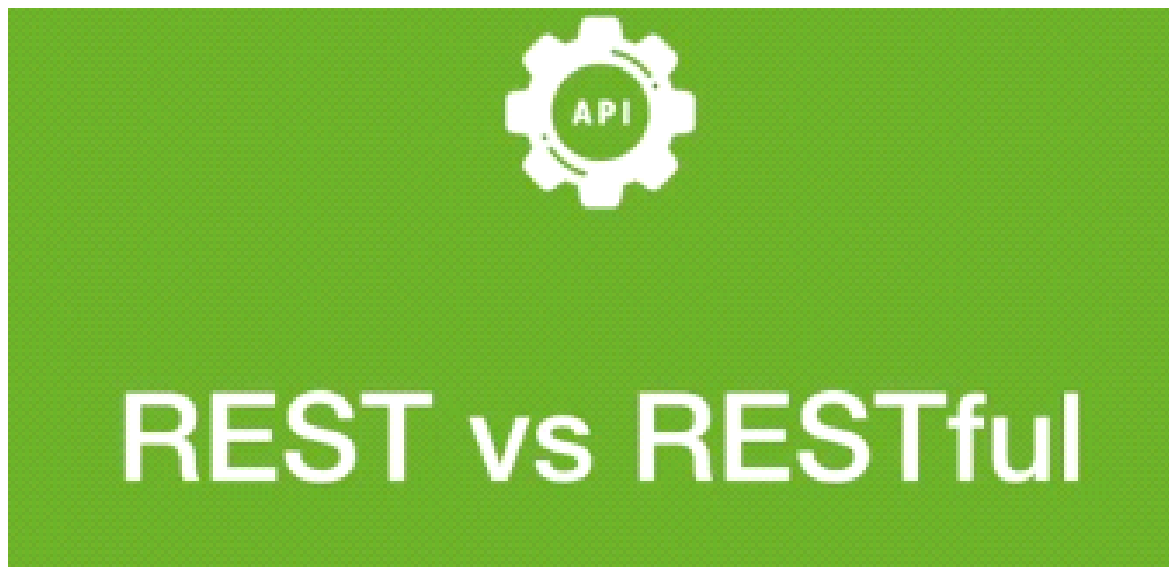
- **Código sob Demanda**

O servidor pode mandar um código para ser executado no lado do cliente, porém, essa constraint é pouco usada.

4.3. Diferença entre REST e RESTful

quarta-feira, 8 de fevereiro de 2023

15:29



Rest é o modelo arquitetural que possui as constraints e RESTful é uma api desenvolvida em conformidade com o modelo REST. Muitos sistemas não são 100% RESTful pois violam alguma constraint definida dentro do modelo. Por isso há diferentes interpretações de uso no mercado

4.4. Desenvolvedores de REST APIs puristas e pragmáticos

quarta-feira, 8 de fevereiro de 2023

15:33



Desenvolvedores puristas aderem 100% as melhores práticas especificadas por Roy Fielding e Pragmáticos são abertos a mudarem as regras do modelo REST para o desenvolvimento de APIS. Esses dois perfis de devs são importantes para se terem discussões, pois para ser mais pragmático, esse perfil deve pensar bem nas melhores estratégias técnicas e não técnicas

4.5. Conhecendo o protocolo HTTP

quarta-feira, 8 de fevereiro de 2023

15:41



Protocolo HTTP

REST APIs independe de linguagem e protocolo mas precisamos escolher tais tecnologias, o protocolo mais comum utilizado é o HTTP. Esse protocolo usa a comunicação com requisições e respostas (Cliente-Servidor) e a **constraint cliente-servidor do REST se define**.

Elementos de uma requisição:

Composição da requisição

```
[MÉTODO] [URI] HTTP/[Versão]
[Cabeçalhos]

[CORPO/PAYLOAD]
```

```
POST /produtos HTTP/1.1
Content-Type: application/json
Accept: application/json

{
  "nome": "Notebook i7",
  "preco": 2100.0
}
```

Os métodos também são chamados de verbos HTTP, essa especificação indica a ação desejada para ser executada. a URI identifica o caminho que queremos utilizar dentro do sistema, a versão do HTTP e o cabçalho que fornece as informações da requisição, por exemplo, o content-type indicando o tipo de conteúdo está sendo enviado no corpo da requisição e a forma de dados que a requisição aceita, se o servidor não poder fornecer o tipo de dado corretamente, é melhor que ocorra um erro. O corpo da

requisição é o dado que estamos enviado para o servidor ou recebemos.

Elementos da Resposta:

Composição da resposta

```
HTTP/[Versão] [STATUS]  
[Cabeçalhos]  
  
[CORPO]
```

```
HTTP/1.1 201 Created  
Content-Type: application/json  
  
{  
  "codigo": 331,  
  "nome": "Notebook i7",  
  "preco": 2100.0  
}
```

O status indica o resultado do processamento da requisição, sendo bem sucedido ou não, indica sempre um resultado para cada tipo de processamento,

4.6. Usando o protocolo HTTP

quarta-feira, 8 de fevereiro de 2023 15:54

4.7. Instalando e testando o Postman

quarta-feira, 8 de fevereiro de 2023 16:30

4.8. Entendendo o que são Recursos REST

quarta-feira, 8 de fevereiro de 2023

16:32



Recursos ou Resources - Qualquer coisa exposta na web que tem importância suficiente para ser representado como **parte do sistema** como, uma nota fiscal, uma lista de clientes, um catálogo...

Tipos de Resources

- Singleton Resource



É uma unidade individual em um recurso, fazendo um paralelo à orientação a objetos, é como se fosse um **objeto de uma classe**. Um recurso único como um objeto produto da classe Produto.

- Collection Resource



Uma coleção de recursos contém **zero ou muitos recursos do mesmo tipo**, uma collection resource.

4.9. Identificando recursos REST

quarta-feira, 8 de fevereiro de 2023

16:56



Identificando Recursos

URI

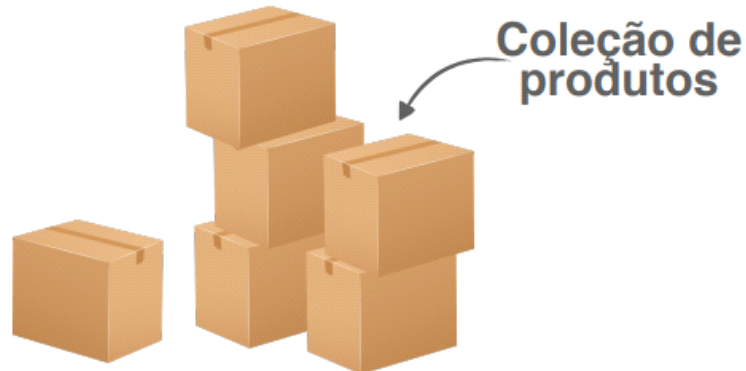
Uniform Resource Identifier

O modelo REST utiliza URI's (Identificador de Recurso Uniforme) para identificar os recursos do sistema, conjunto de caracteres que dão forma a um endereço para determinado recurso no sistema e garantir o acesso a ele ou a coleção de recursos.

URI vs URL

Uma URL é um tipo de URI que significa Uniform Resource Locator ou Localizador de Recurso Uniforme, além de ser uma URI ele também localiza onde o recurso está disponível, onde o recurso está disponível e qual o mecanismo para chegar até ele. ex

/listarProdutos

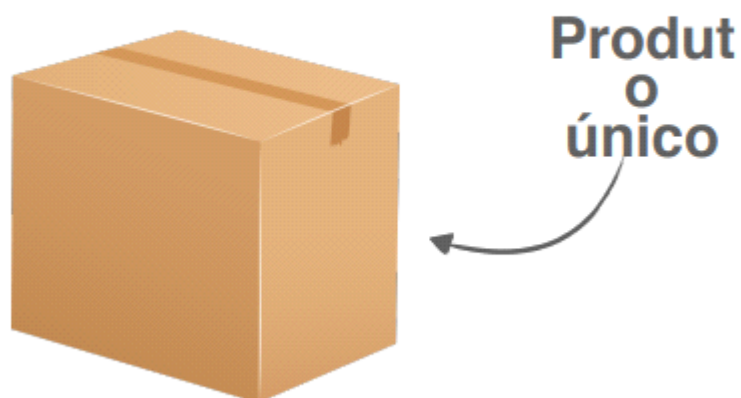


<https://api.algamarket.com.br/listarProdutos>

a URI é somente o endereço do recurso e a URL é o caminho completo para o cliente chegar até o recurso na internet.

Não é uma boa prática identificar um recurso através do verbo/ação e sim através de um substantivo, coisas possuem propriedades, verbos não, seguindo boas práticas, é interessante mudar a URI para `/produtos` mudando apenas os verbos HTTP para acesso ao recurso.

/produtos/{codigo}



<https://api.algamarket.com.br/produtos/331>

O ideal e um consenso no mercado é utilizar os nomes dos recursos no plural quando

for recursos singulares e utilizar o id.

4.10. Modelando e requisitando um Collection Resource com GET

quinta-feira, 9 de fevereiro de 2023 00:24

Criando um controller com o padrão DDD, separando em um pacote de controladores, o controlador será anotado com `@RestController` pois tem em sua declaração a anotação `@Controller` e `@ResponseBody` que são essenciais para a requisição web funcione corretamente.

Definições interessantes:

- Classe anotada com `@RestController` - Uma anotação de conveniência que é anotada com `@Controller` e `@ResponseBody`. Os tipos que carregam essa anotação são tratados como controladores onde os métodos `@RequestMapping` assumem a semântica `@ResponseBody` por padrão. O Spring os chama de meta-anotações ou anotações composta.

```
@RestController
@RequestMapping("/cozinhas")
public class CozinhaController {

    1 usage
    @Autowired
    private CozinhaRepositoryImpl cozinhaRepository;

    @GetMapping
    public List<Cozinha> listar(){
        return cozinhaRepository.todas();
    }
}
```

- `@GetMapping` - Anotação para mapear solicitações HTTP GET em métodos de manipulador específicos. Especificamente, `@GetMapping` é uma anotação composta que atua como um atalho para `@RequestMapping(method = RequestMethod.GET)`.

4.11. Desafio collection resource de estados

quinta-feira, 9 de fevereiro de 2023 09:28

Criar controller de estado

```
@RestController
@RequestMapping("/estados")
public class EstadoController {

    1 usage
    @Autowired
    private EstadoRepository estadoRepository;

    @GetMapping
    public List<Estado> listar(){
        return estadoRepository.todos();
    }
}
```

4.12. Representações de recursos e content negotiation

quinta-feira, 9 de fevereiro de 2023

09:33



Representações de Recursos e Content Negotiation

Um recurso é qualquer coisa exposta na web como um documento, um produto, nota fiscal ou qualquer outra entidade do sistema e um recurso para ser **alcançado** no sistema, precisa ser identificado por uma URI, mais precisamente, precisamos de uma URL para requisitar usando o protocolo HTTP, são coisas diferentes.

O que um URL de um recurso deve retornar?

Representações de Recursos

código que descreve o estado atual de um recurso

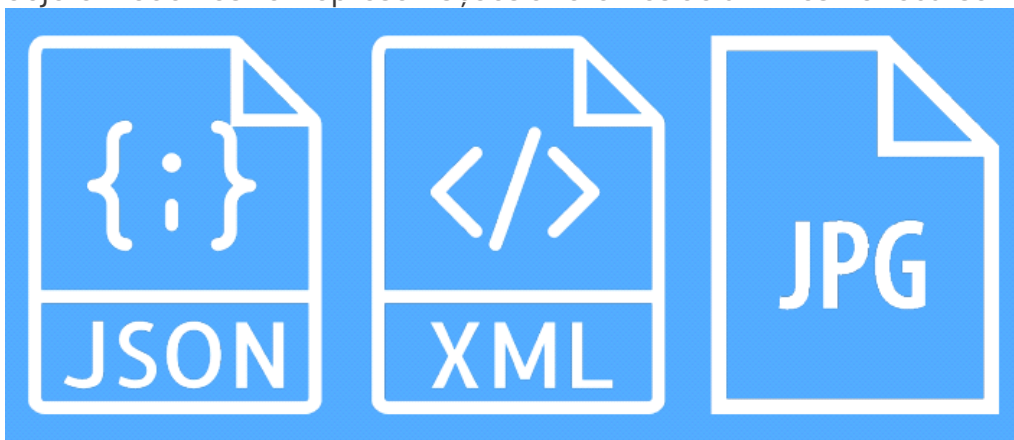
/produtos

```
[
  {
    "codigo": 331,
    "nome": "Notebook i7",
    "preco": 2100.0
  },
  {
    "codigo": 332,
    "nome": "Monitor Dell",
    "preco": 830.0
  },
]
```

JSON



A representação do recurso em JSON que está sendo retornado pela URI não é o objeto em si, conceitualmente, é apenas a representação dele. O recurso é o objeto. Podemos ver representações diferentes de um mesmo recurso



Quando um cliente da API requisitar recursos pela URI ou URL, ele pode especificar qual representação consegue interpretar/aceitar. Essa informação é adicionada no corpo da requisição no **campo/chave Accept** e o valor é considerado um **MediaType**

REQUISIÇÃO

```
GET /produtos HTTP/1.1
Accept: application/json
```

* Logicamente, o servidor terá que ter suporte para esse tipo de representação, caso contrário o servidor negará e enviará um erro.

Outro tipo de MediaType mais comum é o xml

REQUISIÇÃO

```
GET /produtos HTTP/1.1  
Accept: application/xml
```

/produtos

```
<produtos>  
  <produto>  
    <codigo>331</codigo>  
    <nome>Notebook i7</nome>  
    <preco>2100.0</preco>  
  </produto>  
  <produto>  
    <codigo>332</codigo>  
    <nome>Monitor Dell</nome>  
    <preco>830.0</preco>  
  </produto>  
</produtos>
```

XML



Quando um recurso pode oferecer mais de um tipo de representação, chamamos isso de **Content Negotiation**

4.13. Implementando content negotiation para retornar JSON ou XML

quinta-feira, 9 de fevereiro de 2023 10:12

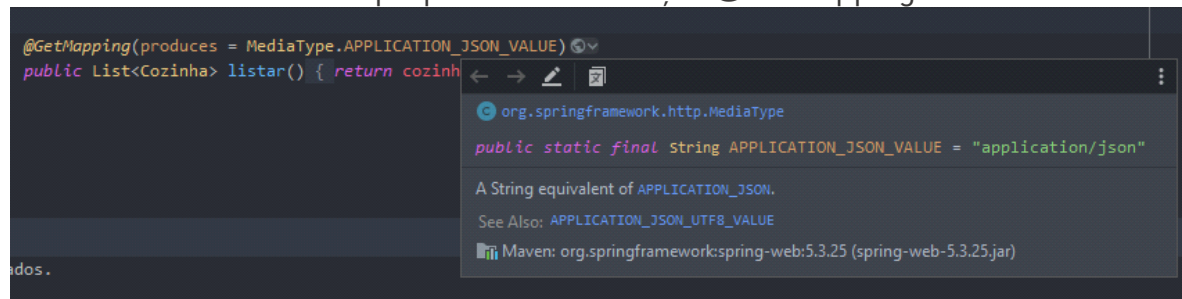
O Spring autoconfigura (por convenção) que a aplicação retornará requisições no formato Json mas isso pode ser configurado e podemos adicionar mais tipos de MediaTypes.

Para que o Spring serialize recursos em um formato XML, é necessário adicionar a dependência do jacksonxml:

```
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
  <version>2.14.2</version>
</dependency>
```

E apenas isso fará com que recursos possam ser representados no formato xml, contando que o cliente especifique o tipo de MediaType está requisitando ao servidor.

Para limitar um recurso a apenas um formato específico para a representação, é necessário adicionar uma propriedade a anotação @GetMapping



Podemos utilizar uma constante que retorna uma String "application/json"

Status: 406 Not Acceptable Time: 98 ms

406 Not Acceptable

The requested resource is only capable of generating content not acceptable according to the Accept headers sent in the request.

Ao chamar a requisição especificando um formato xml, um erro ocorrerá .

Podemos adicionar no escopo da classe a restrição de tipo de retorno do recurso

```
@RestController
@RequestMapping(value = "/cozinhas", produces = MediaType.APPLICATION_JSON_VALUE)
public class CozinhaController {
```

ou em métodos diferentes, a propriedade produces aceita um array de strings

4.14. Consultando Singleton Resource com GET e @PathVariable

quinta-feira, 9 de fevereiro de 2023

10:52

Controller com pathvariable

```
@GetMapping("/{CozinhaId}")
public Cozinha buscar(@PathVariable("CozinhaId") Long id){
    return cozinhaRepository.findById(id);
}
```

O mapeamento da requisição se funde ao mapeamento do método get

```
@RequestMapping(value = "/cozinhas", produces = MediaType.APPLICATION_JSON_VALUE)
public class CozinhaController {
```

ficando /cozinhas/1

para simplificar

```
@GetMapping("/{cozinhaId}")
public Cozinha buscar(@PathVariable Long cozinhaId){
    return cozinhaRepository.findById(cozinhaId);
}
```

O bind será feito normalmente

4.15. Customizando as representações XML e JSON com @JsonIgnore, @JsonProperty e @JsonRootName

quinta-feira, 9 de fevereiro de 2023 11:02

Para customizações na representação do recurso em si, podemos utilizar algumas propriedades que a [biblioteca Jackson](#) fornece

```
@Data
@EqualsAndHashCode(doNotUseGetters = true)
@Entity
@JsonRootName("gastronomia")
@Table(name = "cozinha")
public class Cozinha {

    @EqualsAndHashCode.Include
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)//provedor de persistencia
    private Long id;

    @Column(name = "nome", length = 30, nullable = false)
    @JsonProperty("titulo")
    private String nome;
```

```
<gastronomia>
  <id>1</id>
  <titulo>Tailandesa</titulo>
</gastronomia>

1  {
2    "id": 1,
3    "titulo": "Tailandesa"
4  }
```

ou ignorar totalmente a propriedade na serialização com @JsonIgnore

```
@EqualsAndHashCode(doNotUseGetters = true)
@Entity
@JsonRootName("gastronomia")
@Table(name = "cozinha")
public class Cozinha {

    @EqualsAndHashCode.Include
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)//provedor de persistencia
    private Long id;

    @Column(name = "nome", length = 30, nullable = false)
    @JsonIgnore
    private String nome;
}
```

Para ignorar a nível de classe usamos @JsonIgnoreProperties e passamos as

propriedades a serem ignoradas, para ignorar a nível de propriedade de classe usamos `@JsonIgnore`

Atualmente, estamos usando o próprio recurso para representar em um formato de `MediaType`, porém, há uma discussão sobre isso e forma de abordagens.

4.16. Customizando a representação em XML com Wrapper e anotações do Jackson

quinta-feira, 9 de fevereiro de 2023

19:05

Em uma representação em XML, há um padrão para visualização de listas e entidades, como:

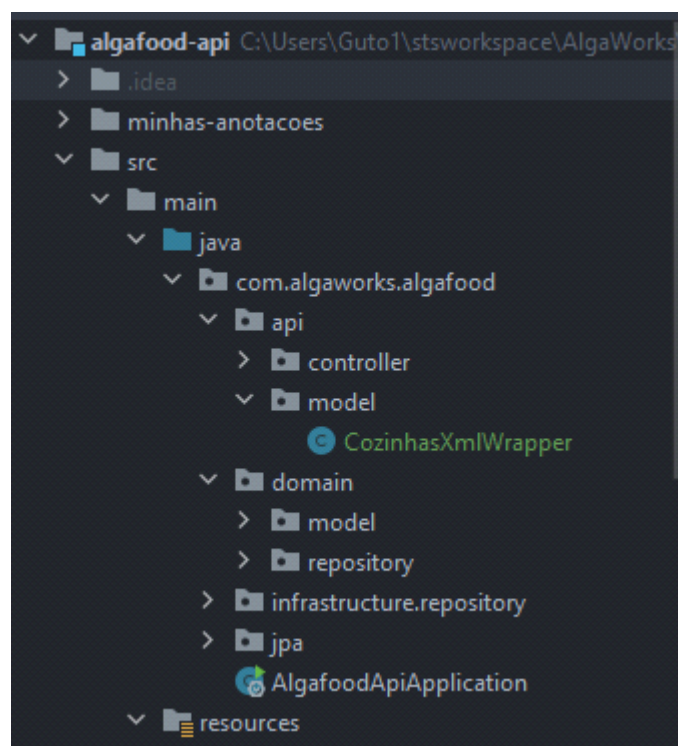


The screenshot shows an XML viewer with tabs for 'Pretty', 'Raw', 'Preview', and 'Visualize'. The 'XML' tab is selected. The XML content is as follows:

```
1 <List>
2   <item>
3     <id>1</id>
4     <nome>Tailandesa</nome>
5   </item>
6   <item>
7     <id>2</id>
8     <nome>Indiana</nome>
9   </item>
10 </List>
```

Um `<List></List>` para representar listas e `<item></item>` para entidades/objetos, sendo possível a modificação dos mesmo.

É necessário criar uma classe para mudar a nomenclatura da lista em `<List>` e criar um modelo de representação da entidade para representar a entidade no formato XML (ou em outro formato se precisar)



Foi criada uma classe para encapsular a lista de cozinhas que será representada no formato XML, declarada com um atributo privado do tipo lista de cozinha que recebe como parâmetro do construtor uma lista de cozinha.

Como a classe servirá como modelo de representação de um recurso, fez sentido ser colocada em um pacote de modelo dentro da camada de api do sistema, pois a classe `CozinhasXmlWrapper` tem um escopo de controlador, e não de domínio do sistema.

```
3 usages
@Data
public class CozinhasXmlWrapper {

    //criando um construtor que recebe no parâmetro uma lista de cozinhas
    @NonNull
    private List<Cozinha> cozinhas;
}
```

`@Data` do Lombok para os getters e setters, `toString` e `EqualsAndHashCode`, para fazer um construtor com um atributo, é necessário anotar a classe com `@NonNull` do Lombok.

```
@GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
public List<Cozinha> listar(){
    return cozinhaRepository.todas();
}

@GetMapping(produces = MediaType.APPLICATION_XML_VALUE)
public CozinhasXmlWrapper listarXml(){
    return new CozinhasXmlWrapper(cozinhaRepository.todas());
}
```

O controlador pode ter mais de um método `Get` sem mapeamento se eles se diferenciarem em algum aspecto, no caso, o `MediaType` aceito no escopo do método.

O retorno da requisição no método `listarXml`

GET http://localhost:8080/cozinhas

Params Authorization Headers (7) Body Pre-request Script Tests

Headers 6 hidden

KEY	VALUE
<input checked="" type="checkbox"/> Accept	application/xml
Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize XML

```
1 <List>
2   <item>
3     <id>1</id>
4     <nome>Tailandesa</nome>
5   </item>
6   <item>
7     <id>2</id>
8     <nome>Indiana</nome>
9   </item>
10 </List>
```

Por padrão, uma coleção de recursos (em xml) vem com o nome `List` e seus elementos com `item`, para alterar é necessário anotar a classe com `@JacksonXmlElement` passando a propriedade `localName` com o nome da lista, que no caso é `cozinhas`.

```
3 usages
@Data
@JacksonXmlElement(localName = "cozinhas")
public class CozinhasXmlWrapper {

    //criando um construtor que recebe no parâmetro uma lista de cozinhas
    @NotNull
    private List<Cozinha> cozinhas;
}
```

```
1 <cozinhas>
2   <cozinhas>
3     <cozinhas>
4       <id>1</id>
5       <nome>Tailandesa</nome>
6     </cozinhas>
7     <cozinhas>
8       <id>2</id>
9       <nome>Indiana</nome>
10    </cozinhas>
11  </cozinhas>
12 </cozinhas>
```

Agora temos 3 níveis de representação com o mesmo nome, e para alterar isso precisamos mudar a propriedade a nível de classe anotado com `@JsonProperty`

```
@Data
@JacksonXmlElement(localName = "cozinhas")
public class CozinhasXmlWrapper {

    //criando um construtor que recebe no parâmetro uma lista de cozinhas
    @NotNull
    @JsonProperty("cozinha")
    private List<Cozinha> cozinhas;
}
```

```
<cozinhas>
  <cozinha>
    <cozinha>
      <id>1</id>
      <nome>Tailandesa</nome>
    </cozinha>
    <cozinha>
      <id>2</id>
      <nome>Indiana</nome>
    </cozinha>
  </cozinha>
</cozinhas>
```

O 1º nível da representação representa a classe anotada com `@JacksonXmlElement` o 2º é uma camada wrapper nas listas, podemos alterar isso removendo a camada wrapper com `@JacksonXmlElementWrapper` com a propriedade `useWrapping` com `false`.

```

@Data
@JacksonXmlRootElement(localName = "cozinhas")
public class CozinhasXmlWrapper {

    //criando um construtor que recebe no parâmetro uma lista de cozinhas
    @NonNull
    @JsonProperty("cozinha")
    @JacksonXmlElementWrapper(useWrapping = false)
    private List<Cozinha> cozinhas;
}

```

Body Cookies Headers (5) Test Results

Pretty

Raw

Preview

Visualize

XML

```

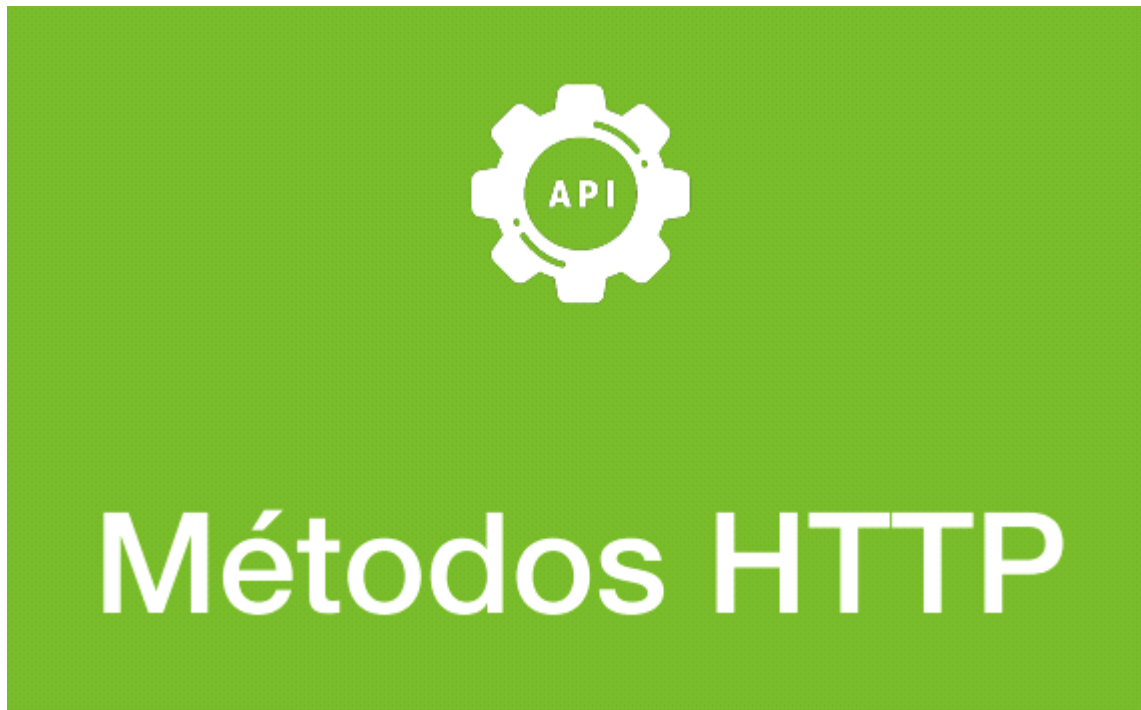
1 <cozinhas>
2   <cozinha>
3     <id>1</id>
4     <nome>Tailandesa</nome>
5   </cozinha>
6   <cozinha>
7     <id>2</id>
8     <nome>Indiana</nome>
9   </cozinha>
10 </cozinhas>

```

4.17. Conhecendo os métodos HTTP

sexta-feira, 10 de fevereiro de 2023

09:25



Composição da requisição

<code>[MÉTODO] [URI] HTTP/[Versão] [Cabeçalhos] [CORPO/PAYLOAD]</code>	<code>POST /produtos HTTP/1.1 Content-Type: application/json Accept: application/json { "nome": "Notebook i7", "preco": 2100.0 }</code>
---	--

Toda requisição HTTP possui um verbo obrigatoriamente, através do método, dizemos ao servidor que tipo de ação queremos realizar no recurso identificado pela URI fornecida.

- **Idempotência**

É a capacidade de aplicar alterações pela primeira vez em algo e repetir as mesmas ações mais vezes sem que o resultado da primeira alteração se altere novamente. Algo idempotente é algo que não muda além da primeira vez.

- MÉTODOS HTTP
 - GET

- É um método idempotente (não gera alteração se for chamado repetidas vezes) e ele não pode ser chamado para modificar recursos, por isso ele é um **método seguro** ou safe method
- POST
 - É usado para criar novos recursos enviando os dados no corpo da requisição, ele não é idempotente pois se fizermos uma requisição post várias vezes, serão adicionados os mesmo dados várias vezes.
- PUT
 - Modifica os recursos a partir de um identificador na URI e no corpo da requisição passamos os dados para serem atualizados, e temos que modificar todo (não parcial), mesmo que seja apenas um atributo, é considerado método não seguro pois modifica um recurso, além de ser idempotente.
o PUT também serve para criar recursos quando não conter o identificador no servidor, porém, tem que ser cuidadoso para utilizar respeitosamente e manter consistente.
- PATCH
 - Usado para atualizar um atributo do recurso (quando o recurso tem muitos atributos) é considerado não seguro e idempotente.
- DELETE
 - Não seguro e idempotente
- HEAD
 - É como o método GET, mas nunca retorna um corpo na resposta da requisição, serve para capturar o cabeçalho e verificar os detalhes da requisição.
- OPTIONS
 - Consulta os métodos disponíveis, traz uma lista de verbos disponíveis de um recurso



Status HTTP

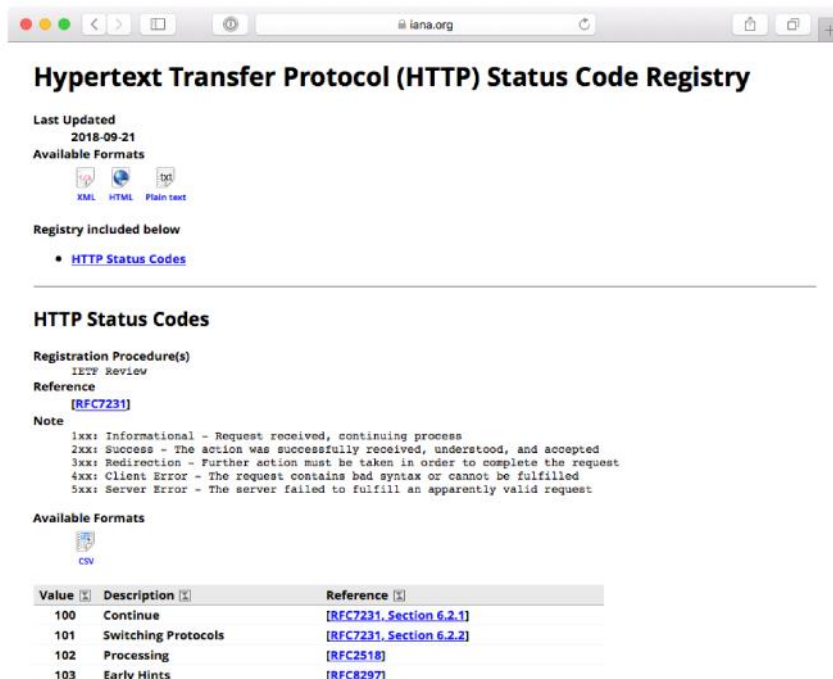
Composição da resposta

HTTP/[Versão] [STATUS]
[Cabeçalhos]

[CORPO]

HTTP/1.1 201 Created
Content-Type: application/json

```
{  
  "codigo": 331,  
  "nome": "Notebook i7",  
  "preco": 2100.0  
}
```



The screenshot shows the IANA website's HTTP Status Code Registry. The page title is "Hypertext Transfer Protocol (HTTP) Status Code Registry". It includes a "Last Updated" date of 2018-09-21 and "Available Formats" for XML, HTML, and Plain text. A link for "HTTP Status Codes" is provided. The "Registration Procedure(s)" section mentions IETF Review and Reference [RFC7231]. A "Note" section lists status code categories: 1xx (Informational), 2xx (Success), 3xx (Redirection), 4xx (Client Error), and 5xx (Server Error). An "Available Formats" section shows a CSV download link. A table lists the first four status codes: 100 (Continue), 101 (Switching Protocols), 102 (Processing), and 103 (Early Hints), each with a reference to an RFC.

Value	Description	Reference
100	Continue	[RFC7231, Section 6.2.1]
101	Switching Protocols	[RFC7231, Section 6.2.2]
102	Processing	[RFC2518]
103	Early Hints	[RFC8297]

Status do HTTP - Nível 200



Status do HTTP - Nível 200



200: OK

Status do HTTP - Nível 200



200: OK

201: Criado

Status do HTTP - Nível 200

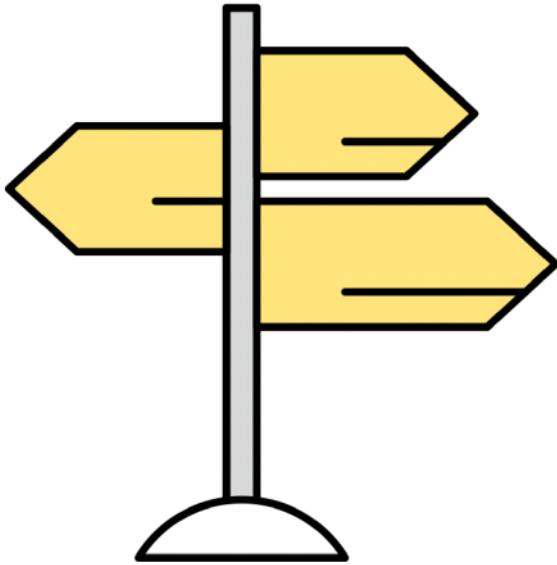


200: OK

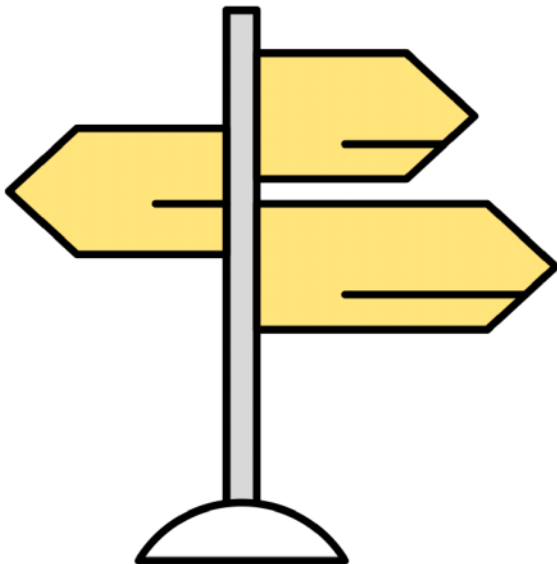
201: Criado

204: Sem conteúdo

Status do HTTP - Nível 300

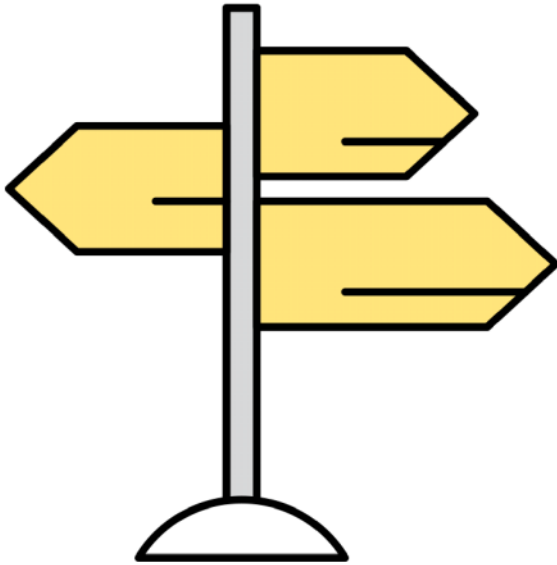


Status do HTTP - Nível 300



301: Movido permanentemente

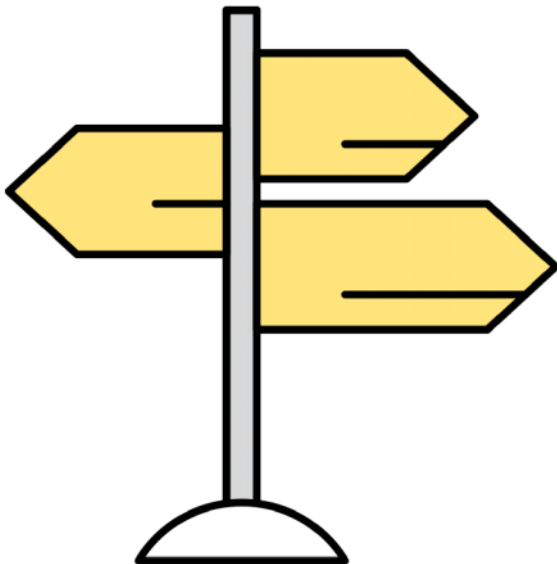
Status do HTTP - Nível 300



301: Movido permanentemente

302: Encontrado

Status do HTTP - Nível 300



301: Movido permanentemente

302: Encontrado

Status do HTTP - Nível 400



Status do HTTP - Nível 400

400: Requisição mal feita



Status do HTTP - Nível 400



400: Requisição mal feita

401: Não autorizado

Status do HTTP - Nível 400



400: Requisição mal feita

401: Não autorizado

403: Proibido

Status do HTTP - Nível 400



400: Requisição mal feita

401: Não autorizado

403: Proibido

404: Não encontrado

Status do HTTP - Nível 400



400: Requisição mal feita

401: Não autorizado

403: Proibido

404: Não encontrado

405: Método não permitido

Status do HTTP - Nível 400



400: Requisição mal feita

401: Não autorizado

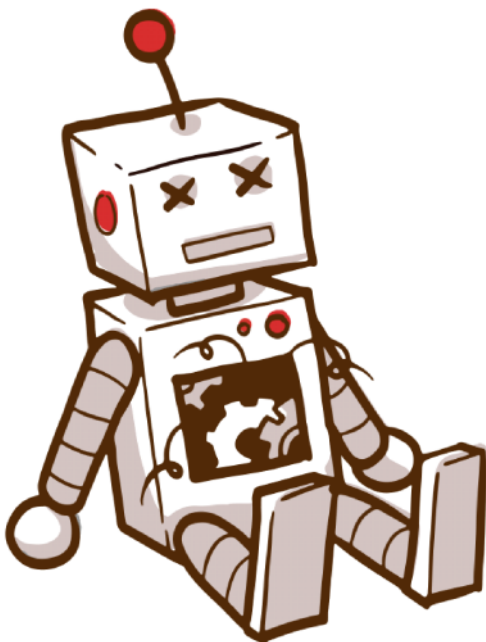
403: Proibido

404: Não encontrado

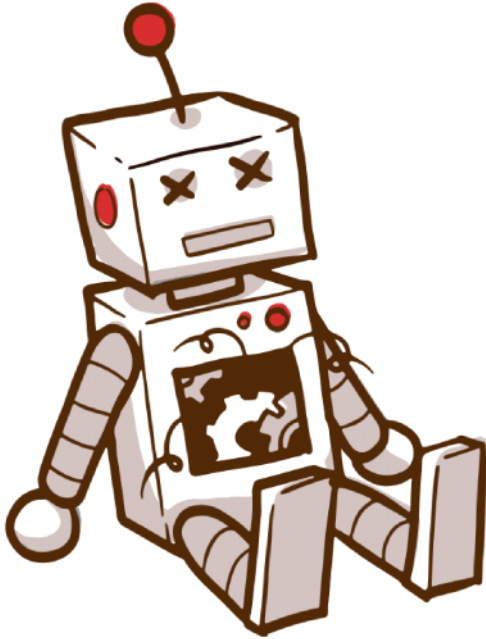
405: Método não permitido

406: Não aceito

Status do HTTP - Nível 500

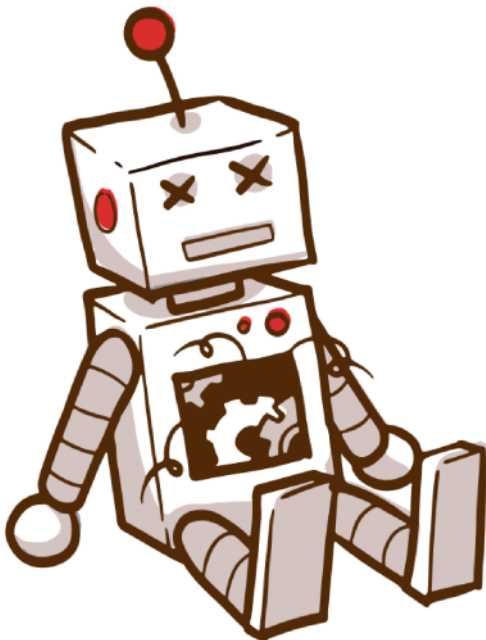


Status do HTTP - Nível 500



500: Erro interno no servidor

Status do HTTP - Nível 500



500: Erro interno no servidor

503: Serviço indisponível

**Quantos códigos de status HTTP
uma API deve usar?**

4.19. Definindo o status da resposta HTTP com @ResponseStatus

sexta-feira, 10 de fevereiro de 2023 14:57

Podemos utilizar a anotação `@ResponseStatus` em um método para customizar o status na resposta da requisição passando como propriedade uma enumeração indicando o tipo de resposta.

```
@GetMapping  
@ResponseStatus(HttpStatus.OK)  
public List<Cozinha> listar() { return cozinhaRepository.todas(); }
```

4.20. Manipulando a resposta HTTP com ResponseEntity

sexta-feira, 10 de fevereiro de 2023 15:04

Atualmente estamos utilizando os recursos como modelo de representação de recurso, ou seja, a biblioteca Jackson serializa o objeto no formato que a requisição solicitar, geralmente em JSON, e retorna no corpo da requisição, mas isso não é considerado uma forma boa de representar. Uma das formas de retorno da requisição é utilizar um `ResponseEntity` e passando um tipo.

```
@GetMapping("/{cozinhaId}")
public ResponseEntity<Cozinha> buscar(@PathVariable Long cozinhaId){
    Cozinha cozinha = cozinhaRepository.findById(cozinhaId);
    |
    return ResponseEntity.status(HttpStatus.OK).body(cozinha);
}
```

ou `ResponseEntity.ok(cozinha);`

```
@GetMapping("/{cozinhaId}")
public ResponseEntity<Cozinha> buscar(@PathVariable Long cozinhaId){
    Cozinha cozinha = cozinhaRepository.findById(cozinhaId);

    HttpHeaders header = new HttpHeaders();
    header.add(HttpHeaders.LOCATION, headerValue: "http://localhost:8080/new/cozinhas");

    return ResponseEntity.status(HttpStatus.FOUND).headers(header).body(cozinha);
}
```

Construção da requisição adicionando um header, um status e um corpo.

DETALHE: O código de resposta de status de redirecionamento HTTP (HyperText Transfer Protocol) 302 **Found** indica que o recurso solicitado foi movido temporariamente para a URL fornecida pelo **Location** no cabeçalho., Ao encontrar uma requisição com Found e uma url com a propriedade, ele automaticamente redireciona

4.21. Corrigindo o Status HTTP para resource inexistente

sexta-feira, 10 de fevereiro de 2023 20:02

Dica rápida para retornar uma resposta adequada a requisição quando um recurso for inexistente

```
@GetMapping("/{cozinhaId}")
public ResponseEntity<Cozinha> buscar(@PathVariable Long cozinhaId){
    Cozinha cozinha = cozinhaRepository.findById(cozinhaId);
    if(cozinha != null)
        return ResponseEntity.ok(cozinha);
    return ResponseEntity.notFound().build();
}
```

Usamos o 204, quando vamos fazer alguma operação que não depende de nenhuma resposta, apenas precisamos saber se a requisição foi finalizada com sucesso.

Por exemplo, no caso da deleção. O cliente que desejar remover um recurso específico, não precisa de nenhum dado no corpo da resposta. O status code é o suficiente para ele saber se deu certo ou não.

Por isso utilizamos o 204. Sempre que uma requisição dessa for concluída com sucesso, não terá nenhum dado para retornar no corpo da resposta.

Agora, quando estamos buscando um recurso e o mesmo não existe, precisamos informar isso ao cliente. Fazemos isso com o status 404.

Por exemplo, se não existir o recurso "/cozinhas", então o correto é o cliente saber disso, com 404.

Se existir "/cozinhas" mas não existir a cozinha 1, por exemplo (ficando "/cozinhas/1") quer dizer que o recurso 1 não existe, logo 404.

4.22. Status HTTP para collection resource vazia qual usar

sexta-feira, 10 de fevereiro de 2023 23:28

Quando retorna uma lista vazia, o correto é utilizar um código de 200 OK pois há um recurso

4.23. Modelando e implementando a inclusão de recursos com POST

sexta-feira, 10 de fevereiro de 2023 23:38

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public void adicionar(@RequestBody Cozinha cozinha){
    cozinhaRepository.adicionar(cozinha);
}
```

A anotação `@RequestBody` significa que a requisição (request) tem um corpo, e o próprio framework faz a instanciação de um objeto a partir dos dados no corpo da requisição.

The screenshot shows a REST client interface. At the top, the method is set to **POST** and the URL is `http://localhost:8080/cozinhas`. Below this, there are tabs for **Params**, **Authorization**, **Headers (8)**, and **Body**. The **Body** tab is selected and highlighted with a red underline. Under the **Body** tab, there are three radio buttons: **none**, **form-data**, and **x-www-form-urlencoded**. The **none** option is selected. Below the radio buttons, there is a text area containing a JSON body:

```
1 {
2   ... "nome" : "africana"
3 }
```

Um status mais apropriado ao utilizar uma requisição de adição com o verbo POST é o status **201 CREATED**

4.24. Negociando o media type do payload do POST com Content-Type

sábado, 11 de fevereiro de 2023 09:25

Na aula [4.13. Implementando content negotiation para retornar JSON ou XML](#) implementamos opções para o retorno da requisição, esse é o `content negotiation`, a capacidade do cliente escolher a o formato da representação do recurso baseado no formato (geralmente, xml ou json) definindo no cabeçalho a chave `accept` com o valor a ser requerido (application/json ou application/xml).

Podemos também definir o formato da representação a ser enviada com `content-type` através do verbo POST

4.25. Modelando e implementando a atualização de recursos com PUT

sábado, 11 de fevereiro de 2023

09:35

```
@PutMapping("/{cozinhaId}")
public ResponseEntity<Cozinha> atualizar(@RequestBody Cozinha cozinha, @PathVariable Long cozinhaId) {
    Cozinha target = cozinhaRepository.findById(cozinhaId);

    if(target != null){
        BeanUtils.copyProperties(cozinha, target, ...ignoreProperties: "id");
        target = cozinhaRepository.adicionar(target);
        return ResponseEntity.ok(target);
    }
    return ResponseEntity.notFound().build();
}
```

O verbo HTTP para atualizar é o PUT e no Spring é sinalizado com a anotação `@PutMapping`, e mapeando o id do recurso pelo parâmetro da requisição. A classe utilitária `BeanUtils` tem um método que copia as propriedades de um objeto e passa para um outro objeto. O objeto `target` é o objeto vindo do banco de dados através do id fornecido na requisição por `@PathVariable`, no método `copyProperties` o 1º argumento é o objeto de origem (`@RequestBody`) e o 2º argumento é o objeto que vai receber as propriedades vindas do 1º argumento, a string fornecida no 3º argumento é a propriedade ignorada, no caso, `copyProperties` vai ignorar o id de `target` e vai permanecer intacto.

- Documentação de `copyProperties`: Copie os valores de propriedade do bean de origem fornecido para o bean de destino especificado, ignorando as "ignoreProperties" fornecidas. Observação: as classes de origem e destino não precisam corresponder ou mesmo ser derivadas uma da outra, desde que as propriedades correspondam.

4.26. Modelando e implementando a exclusão de recursos com DELETE

sábado, 11 de fevereiro de 2023

17:39

```
@DeleteMapping("/{cozinhaId}")
public ResponseEntity<Cozinha> deletar(@PathVariable Long cozinhaId){
    Cozinha target = cozinhaRepository.findById(cozinhaId);

    try {
        if(target != null){
            cozinhaRepository.remove(target);
            return ResponseEntity.noContent().build();
        }
    } catch(DataIntegrityViolationException e){
        return ResponseEntity.status(HttpStatus.CONFLICT).build();
    }

    return ResponseEntity.notFound().build();
}
```

4.27. Implementando a camada de domain services (e a importância da linguagem ubíqua)

sábado, 11 de fevereiro de 2023

17:56

No DDD, existe um conceito chamado domain service ou serviço de domínio, é uma camada considerada "sem estado" que realiza operações específicas a nível de domínio, ou seja uma tarefa de negócio. Quando um processo no domínio não é uma responsabilidade natural de uma entidade, criamos uma camada de serviço, por exemplo, a camada controladora de requisições ter acesso a camada de persistência como a instância de um repositório, há algumas exceções para isso como usar um repositório no controlador para fazer consultas.

Portanto, uma abordagem ideal é separar em uma camada de serviço de domínio, criando uma classe - de serviço de domínio - que se encarrega de tratar processos de responsabilidades com características de fora do escopo de controladores ou repositórios.

```
2 usages
@Service
public class CozinhaService {

    1 usage
    @Autowired
    private CozinhaRepository cozinhaRepository;

    1 usage
    public Cozinha salvar(Cozinha cozinha){

        return cozinhaRepository.salvar(cozinha);
    }
}
```

A classe de serviço poderia ser anotada com @Component mas existe uma anotação pensada no DDD para tornar a classe mais apropriada no sentido semântico de gerenciamento de beans. A linguagem obliqua é utilizada na modelagem do sistema para ter mais proximidade com o negócio da empresa ou da finalidade do sistema, como é um restaurante, o nome mais apropriado no contexto foi "cozinha" mas devemos pensar em outros contextos.

4.28. Refatorando a exclusão de cozinhas para usar domain services

domingo, 12 de fevereiro de 2023 09:14

```
@DeleteMapping("/{cozinhaId}")
public ResponseEntity<Cozinha> deletar(@PathVariable Long cozinhaId) {
    try {
        cozinhaService.excluir(cozinhaId);
        return ResponseEntity.noContent().build();
    } catch (EntidadeEmUsoException e) {
        return ResponseEntity.status(HttpStatus.CONFLICT).build();
    } catch (EntidadeNaoEncontradaException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
    }
}
```

Agora o método de deleção chama os métodos do serviço de domínio e trata as exceções para retornar as requisições adequadas para cada exceção.

```
public void excluir (Long cozinhaId){
    try {
        cozinhaRepository.excluir(cozinhaId);
    } catch (DataIntegrityViolationException e){
        throw new EntidadeEmUsoException(String.format("Cozinha de código %d não pôde ser excluída, pois está em uso",
            cozinhaId));
    }
    catch (EmptyResultDataAccessException e){
        throw new EntidadeNaoEncontradaException(String.format("Cozinha de código %d não pôde ser encontrada",
            cozinhaId));
    }
}
```

O método excluir da camada de serviço de domínio captura as exceções, traduz e relança para o controlador, a camada não tem que ter conhecimento da camada controladora, ou seja, não pode ter classes que fazem parte do controller (retornar um ResponseEntity)

```
public class EntidadeNaoEncontradaException extends RuntimeException {
    1 usage
    public EntidadeNaoEncontradaException(String msg) {
        super(msg);
    }
}
```

As classes Exception estendem RuntimeException

4.29. Desafio modelando e implementando a consulta de recursos de restaurantes

domingo, 12 de fevereiro de 2023 10:30

4.30. Modelando e implementando a inclusão de recursos de restaurantes

domingo, 12 de fevereiro de 2023

15:36

```
public Restaurante salvar(Restaurante restaurante) {  
    Long cozinhaId = restaurante.getCozinha().getId();  
    Cozinha cozinha = cozinhaRepository.findById(cozinhaId);  
  
    if(cozinha == null){  
        throw new EntidadeNaoEncontradaException(String.format("Cozinha de código %d não pôde ser encontrada",  
            cozinhaId));  
    }  
  
    restaurante.setCozinha(cozinha);  
    return restauranteRepository.salvar(restaurante);  
}
```

```
@PostMapping  
public ResponseEntity<?> adicionar(@RequestBody Restaurante restaurante){  
    try{  
        Restaurante restauranteAdd = restauranteService.salvar(restaurante);  
        return ResponseEntity.status(HttpStatus.CREATED).body(restauranteAdd);  
    } catch(EntidadeNaoEncontradaException e){  
        return ResponseEntity.badRequest().body(e.getMessage());  
    }  
}
```

Utilizando Wildcard e retornando um HTTP 400 Bad Request que indica que o servidor não pode ou não irá processar a requisição devido a alguma coisa que foi entendida como um erro do cliente (por exemplo, sintaxe de requisição mal formada, enquadramento de mensagem de requisição inválida ou requisição de roteamento enganosa).

4.31. Desafio Modelando e implementando a atualização de recursos de restaurantes

terça-feira, 14 de fevereiro de 2023

10:25

4.32. Desafio implementando serviços REST de cidades e estados

terça-feira, 14 de fevereiro de 2023 10:25

4.33. Analisando solução para atualização parcial de recursos com PATCH

terça-feira, 14 de fevereiro de 2023 10:25

```
@PatchMapping("/{restauranteId}")
public ResponseEntity<?> atualizarParcial(@PathVariable Long restauranteId,
    @RequestBody Map<String, Object> campos) {
    Restaurante restauranteAtual = restauranteRepository.buscar(restauranteId);

    if (restauranteAtual == null) {
        return ResponseEntity.notFound().build();
    }

    merge(campos, restauranteAtual);

    return atualizar(restauranteId, restauranteAtual);
}

private void merge(Map<String, Object> camposOrigem, Restaurante restauranteDestino) {
    camposOrigem.forEach((nomePropriedade, valorPropriedade) -> {
        System.out.println(nomePropriedade + " = " + valorPropriedade);
    });
}
```

A solução é trazer um map dos valores da requisição de chave-valor, a anotação `@RequestBody` consegue atribuir os elementos chave-valor JSON para uma variável map de `<String, Object>` e percorrer todos os as chaves que foram passadas para o map

4.34. Finalizando a atualização parcial com a API de Reflections do Spring

terça-feira, 14 de fevereiro de 2023 10:30

```
private static void merge(Map<String, Object> dadosOrigem, Restaurante restauranteDestino) {
    dadosOrigem.forEach((nomePropriedade, valorPropriedade) -> {

        ObjectMapper objectMapper = new ObjectMapper();
        final Restaurante restauranteOrigem = objectMapper.convertValue(dadosOrigem, Restaurante.class);

        System.out.println(nomePropriedade + " = " + valorPropriedade);

        Field field = ReflectionUtils.findField(Restaurante.class, nomePropriedade);
        field.setAccessible(true);

        final Object novoValorConvertido = ReflectionUtils.getField(field, restauranteOrigem);

        ReflectionUtils.setField(field, restauranteDestino, novoValorConvertido);
    });
}
```

o método merge recebe os dados da requisição em um map<String, Object> e são somente os campos a serem atualizados parcialmente.

é necessário ter uma instância de ObjectMapper para transformar os elementos chave-valor para propriedade-valor de um objeto Restaurante

a classe utilitária ReflectionUtils.findField aceita no 1º argumento uma classe onde vai procurar a propriedade, no 2º argumento é a string equivalente ao campo para procurar na classe do 1º argumento. A partir disso o campo é adicionado na variável field do tipo Field, por padrão, se o atributo for privado, o field também vai ser privado, logo é necessário deixar acessível o campo com setAccessible.

ReflectionUtils.getField recebe no 1º argumento o field do atributo e no 2º argumento recebe a classe para se obter o valor da propriedade equivalente a field do 1º argumento, retornando um objeto que contém o valor da propriedade

ReflectionUtils.setField recebe no 1º argumento o field para encontrar, 2º argumento a instância para setar no field do 1º argumento, e 3º argumento o valor para setar no atributo field do 2º argumento

Isso é feito para garantir o tipo das propriedades das classes, pois o método setField não consegue diferenciar Integer de Double, ou BigDecimal

4.35. Introdução ao Modelo de Maturidade de Richardson (RMM)

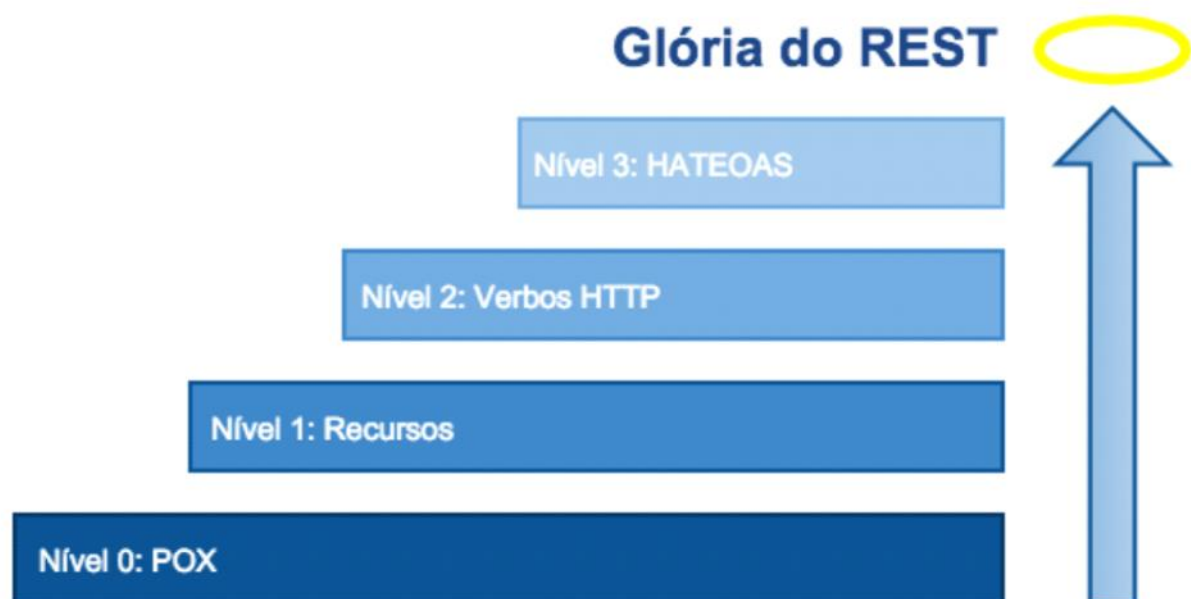
terça-feira, 14 de fevereiro de 2023

10:53



Apesar de Roy Fielding deixar claro que para que uma aplicação seja considerada REST ela precisa seguir todas as constraints definidas no padrão arquitetural REST, mas na prática o mercado não implementa todas as constraints de REST, por falta de conhecimento ou por praticidade.

Os programadores puristas não consideram essas APIS REST, mas somente apis transportando dados, os pragmáticos consideram essas apis REST com base em níveis



Baseado nesse cenário, o programador Leonard Richardson mapeou os padrões de API no mercado, e decidiu nomear de modelo de maturidade de Richardson, que nivela os padrões de API de acordo com a forma com que foi implementado, o mercado está majoritariamente implementando apis no nível 2

4.36. Conhecendo o nível 0 do RMM

terça-feira, 14 de fevereiro de 2023

11:00

Nível 0 - POX (Plain Old XML)

REQUISIÇÃO

```
POST /servicoLoja HTTP/1.1
```

```
<cadastrarProduto>
  <nome>Macbook Pro 13</nome>
  <preco>15000</preco>
</cadastrarProduto>
```

REQUISIÇÃO

```
POST /servicoLoja HTTP/1.1
```

```
<alterarProduto>
  <codigo>984</codigo>
  <nome>Macbook Pro 15</nome>
  <preco>17000</preco>
</alterarProduto>
```

Nível 0 - POX (Plain Old XML)

REQUISIÇÃO

```
POST /servicoLoja HTTP/1.1
```

```
{
  "funcao": "cadastrarProduto",
  "dados": {
    "nome": "Macbook Pro 13",
    "preco": 15000
  }
}
```

Nível 0 - POX (Plain Old XML)

RESPOSTA

HTTP/1.1 200 OK

```
<produto>
  <codigo>984</codigo>
  <nome>Macbook Pro 15</nome>
  <preco>17000</preco>
</produto>
```

RESPOSTA

HTTP/1.1 200 OK

```
<erroCadastroProduto>
  <codigo>984</codigo>
  <nome>Macbook Pro 15</nome>
  <motivo>Não encontrado</motivo>
</erroCadastroProduto>
```

Nível mais rudimentar, e geralmente utiliza um único endpoint, não utiliza os verbos adequadamente, nem status. Utiliza modelo RPC, somente para transportar dados.

4.37 - Conhecendo o nível 1 do RMM

terça-feira, 14 de fevereiro de 2023 11:02

Nível 1 - Recursos

REQUISIÇÃO

```
POST /produtos HTTP/1.1
```

```
<cadastrarProduto>  
  <nome>Macbook Pro 13</nome>  
  <preco>15000</preco>  
</cadastrarProduto>
```

REQUISIÇÃO

```
POST /produtos/984 HTTP/1.1
```

```
<alterarProduto>  
  <nome>Macbook Pro 13</nome>  
  <preco>15000</preco>  
</alterarProduto>
```

No nível 1, ainda não se usam corretamente os verbos HTTP e códigos de status HTTP, mas já identificam recursos específicos na URI da do ENDPOINT

4.38. Conhecendo o nível 2 do RMM

terça-feira, 14 de fevereiro de 2023

11:05

Nível 2 - Verbos HTTP

REQUISIÇÃO

```
POST /produtos HTTP/1.1
```

```
<produto>  
  <nome>Macbook Pro 13</nome>  
  <preco>15000</preco>  
</produto>
```

REQUISIÇÃO

```
PUT /produtos/984 HTTP/1.1
```

```
<produto>  
  <nome>Macbook Pro 13</nome>  
  <preco>15000</preco>  
</produto>
```

Nível 2 - Verbos HTTP

REQUISIÇÃO

```
POST /produtos HTTP/1.1
```

```
{  
  "nome": "Macbook Pro 13",  
  "preco": 15000  
}
```

REQUISIÇÃO

```
PUT /produtos/984 HTTP/1.1
```

```
{  
  "nome": "Macbook Pro 15",  
  "preco": 17000  
}
```


Nível 2 - Verbos HTTP

RESPOSTA

HTTP/1.1 201 Created

```
{  
  "id": 987,  
  "nome": "Macbook Pro 13",  
  "preco": 15000  
}
```

RESPOSTA

HTTP/1.1 404 Not Found

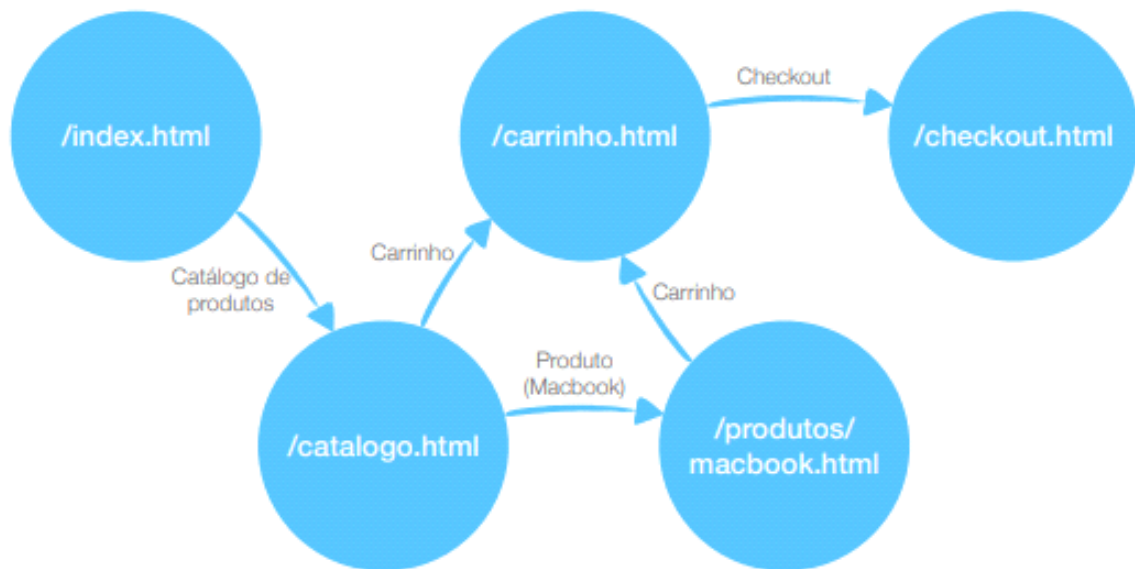
A maioria das aplicações do mercado estão nesse nível e já trabalha com os verbos HTTP corretamente de acordo com a semântica da requisição junto com URIS definidas para cada tipo de requisição. Os códigos de resposta também estão nesse nível

4.39. Conhecendo o nível 3 do RMM

terça-feira, 14 de fevereiro de 2023

11:27

Nível 3 - HATEOAS



O nível 3 é denominado HATEOAS - Hypertext As The Engine Of Application State - Hipertexto como motor de estado da aplicação.

O proposito central é dizer qual o próximo passo ao acessar um recurso, como em uma navegação de páginas web

Nível 3 - HATEOAS

REQUISIÇÃO

GET `/produtos/984`

REQUISIÇÃO

HTTP/1.1 200 OK

```
{
  "id": 73,
  "nome": "Macbook Pro 13",
  "preco": 15000,
  "links": {
    "inativar": "/pagamentos/73",
    "fornecedor": "/fornecedores/34"
  }
}
```