

FIAP GRADUAÇÃO

DIGITAL BUSINESS ENABLEMENT

Prof. THIAGO T. I. YAMAMOTO

#06 – WEB SERVICES RESTFUL



thiagoyama



thiagoyama@gmail.com

- Web services RESTFul
- JSON
- Exemplos:
- GET
- POST
- PUT
- DELETE

Webservice: integração entre sistemas diferentes;

RESTFul (REpresentational State Transfer)

- Simples, leve, fácil de desenvolver e evoluir;
- Tudo é um recurso (Resource);
- Cada recurso possui um identificador (URI);
- Recursos podem ser de vários formatos: html, xml, json;
- Protocolo HTTP;
- Os métodos HTTP: GET, POST, PUT, DELETE são utilizados na arquitetura REST.

- **GET:** recupera informações de um recurso;
- **POST:** cria um novo recurso;
- **PUT:** atualiza um recurso;
- **DELETE:** remove um recurso;

Quando realizamos uma requisição, precisamos falar o caminho da mesma:

POST	/pedido/	Criar
GET	/pedido/1	Visualizar
PUT	/pedido/1	Alterar
DELETE	/pedido/1	Apagar

WS RESTFUL - JAVA

JAX-RS

- Especificação Java para suporte a REST (JSR 331)
- JAX-RS: Java API for RESTful Web Services
- **Jersey**: implementação da especificação.

 Jersey - RESTful Web Services in Java.

 **Jersey**

<https://jersey.java.net/>

RESTful Web Services in Java.

About

Developing RESTful Web services that seamlessly support exposing your data in a variety of representation media types and abstract away the low-level details of the client-server communication is not an easy task without a good toolkit. In order to simplify development of RESTful Web services and their clients in Java, a standard and portable [JAX-RS API](#) has been designed. Jersey RESTful Web Services framework is open source, production quality, framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation.

Jersey framework is more than the JAX-RS Reference Implementation. Jersey provides it's own [API](#) that extend the JAX-RS toolkit with additional features and utilities to further simplify RESTful service and client development. Jersey also exposes numerous extension SPIs so that developers may extend Jersey to best suit their needs.

Principais anotações:

@Path()	Define o caminho para o recurso (URI).
@POST	Responde por requisições POST.
@GET	Responde por requisições GET.
@PUT	Responde por requisições PUT.
@DELETE	Responde por requisições DELETE.
@Produces()	Define o tipo de informação que o recurso retorna.
@Consumes()	Define o tipo de informação que o recurso recebe.
@PathParam	Injeta um parâmetro da URL no parâmetro do método.

CRIANDO O PROJETO

New Dynamic Web Project

Dynamic Web Project
Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name:

Project location
☒ Use default location
Location:

Target runtime

Dynamic web module version

Configuration

A good starting point for working with Apache Tomcat v9.0 runtime. Additional facets can later be installed to add new functionality to the project.

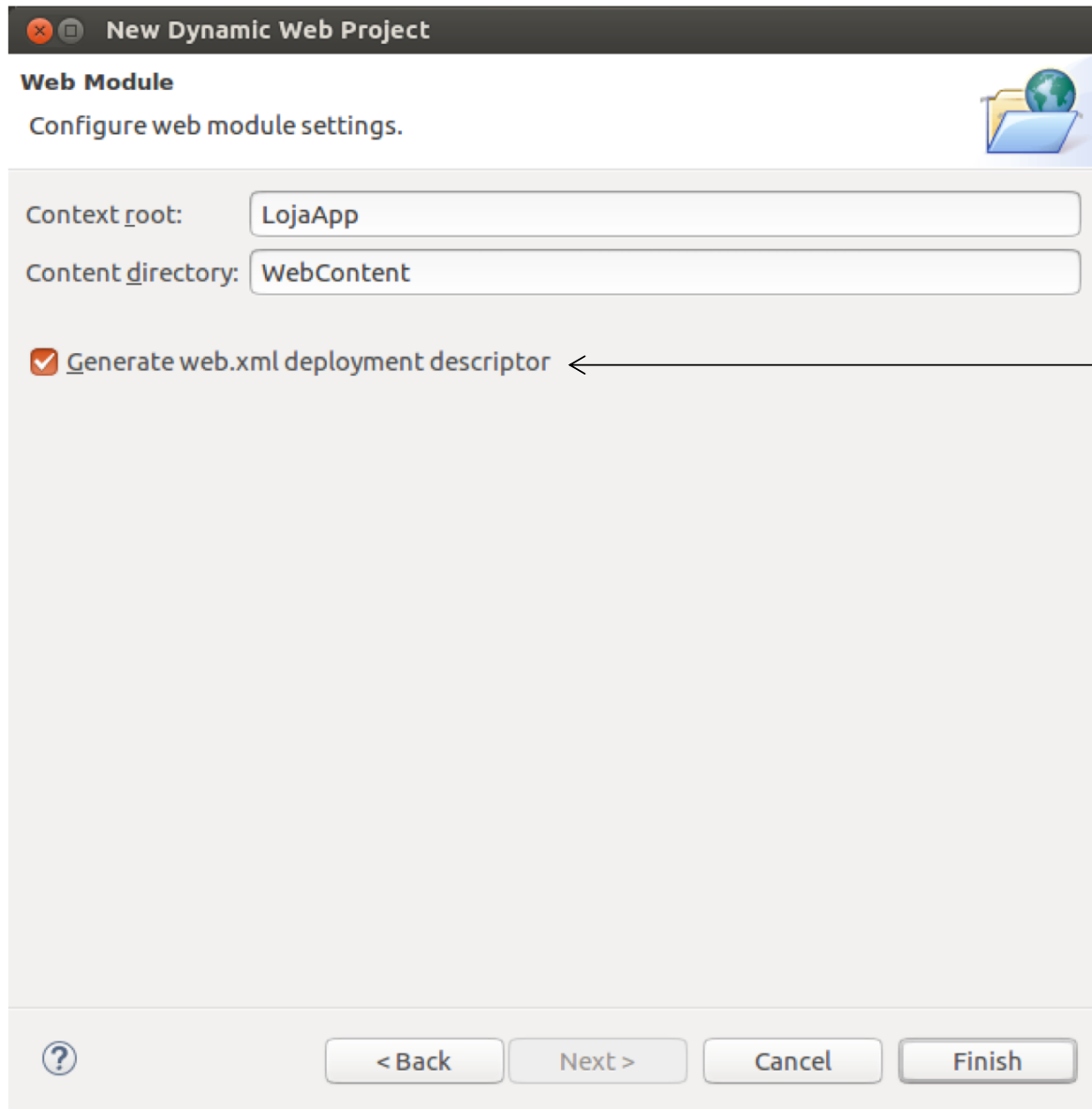
EAR membership
☐ Add project to an EAR

Crie um Dynamic Web Project com **Tomcat** e **web.xml**

Module Version 3.1

Next para gerar o web.xml

❑ CRIANDO O PROJETO



New Dynamic Web Project

Web Module
Configure web module settings.

Context root: LojaApp

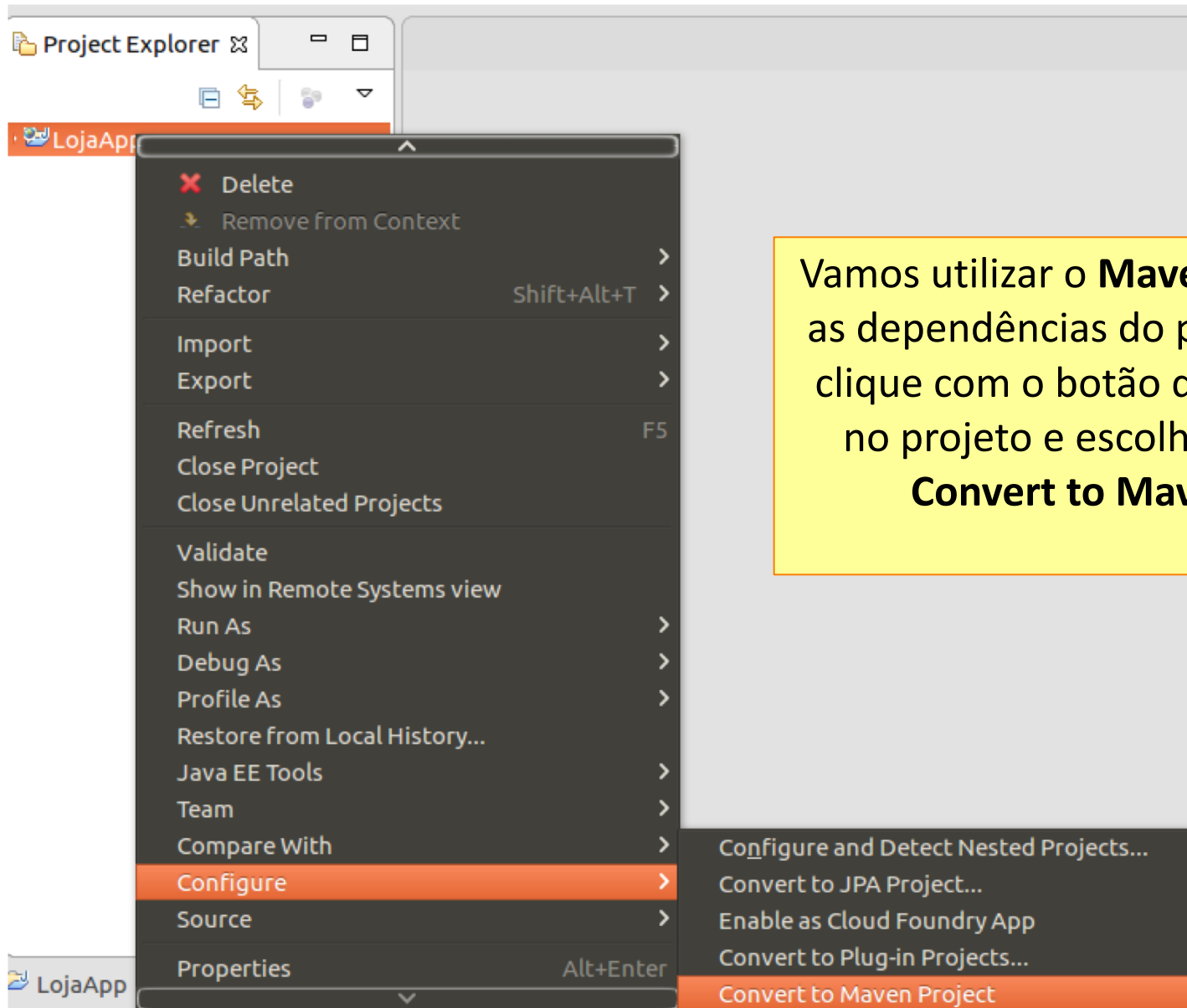
Content directory: WebContent

☒ Generate web.xml deployment descriptor <

? < Back Next > Cancel Finish

Marque para gerar o
web.xml

CONFIGURANDO O PROJETO



Vamos utilizar o **Maven** para gerenciar as dependências do projeto, para isso clique com o botão direito do mouse no projeto e escolha **Configure** → **Convert to Maven Project**

Maven é uma ferramenta para o gerenciamento, construção e implantação de projetos Java. Com ele é possível gerenciar as dependências, o build e documentação.

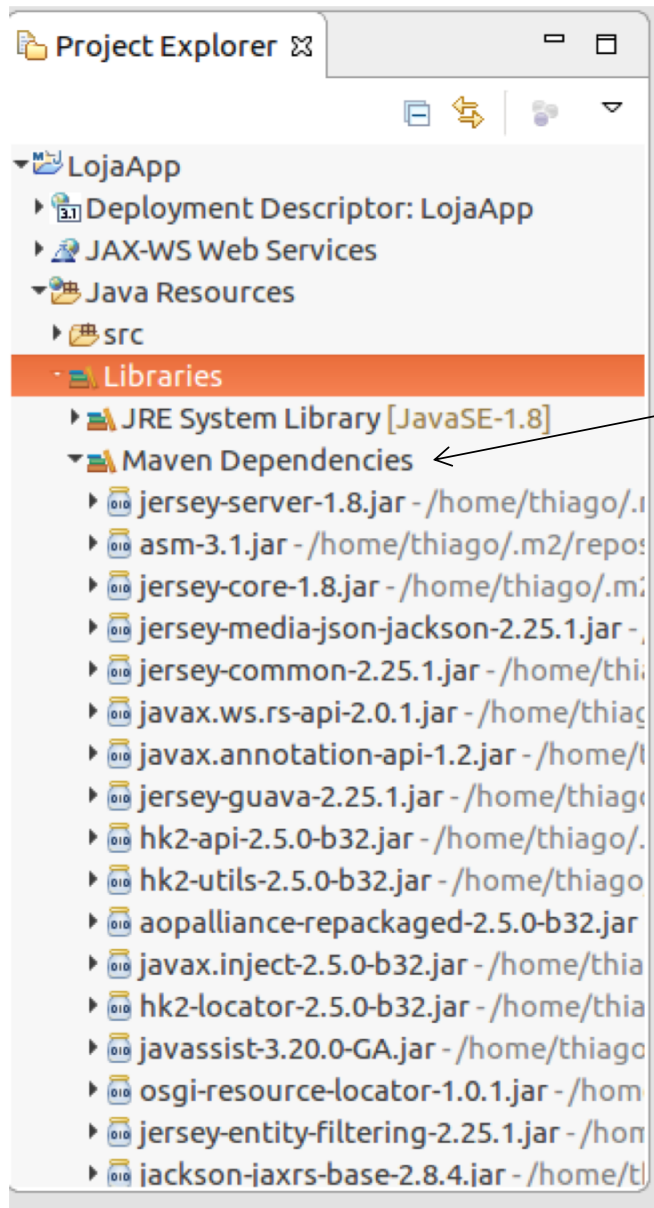
O arquivo **pom.xml** deve ficar na raiz do projeto e nele se declara a estrutura, dependências e características do seu projeto.

Vamos configurar o pom.xml para adicionar as dependências do projeto:

```
<dependencies>
  <dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-server</artifactId>
    <version>2.17</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-servlet-core</artifactId>
    <version>2.17</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>2.25.1</version>
  </dependency>
</dependencies>
```

Adicione as dependências após
a tag </build>

MAVEN DEPENDENCIES



Após a configuração clique com o botão direito do mouse no projeto e escolha **Maven**
→ **Update Project**

Depois é possível ver as bibliotecas (jar) que foram adicionadas ao projeto

Agora é preciso configurar o projeto para o Restful, no arquivo web.xml adicione:

```
<servlet>
```

```
  <servlet-name>jersey-servlet</servlet-name>
```

```
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
```

```
  <init-param>
```

```
    <param-name>jersey.config.server.provider.packages</param-name>
```

```
    <param-value>br.com.fiap.resource</param-value>
```

```
  </init-param>
```

```
  <init-param>
```

```
    <param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>
```

```
    <param-value>>true</param-value>
```

```
  </init-param>
```

```
  <load-on-startup>1</load-on-startup>
```

```
</servlet>
```

```
<servlet-mapping>
```

```
  <servlet-name>jersey-servlet</servlet-name>
```

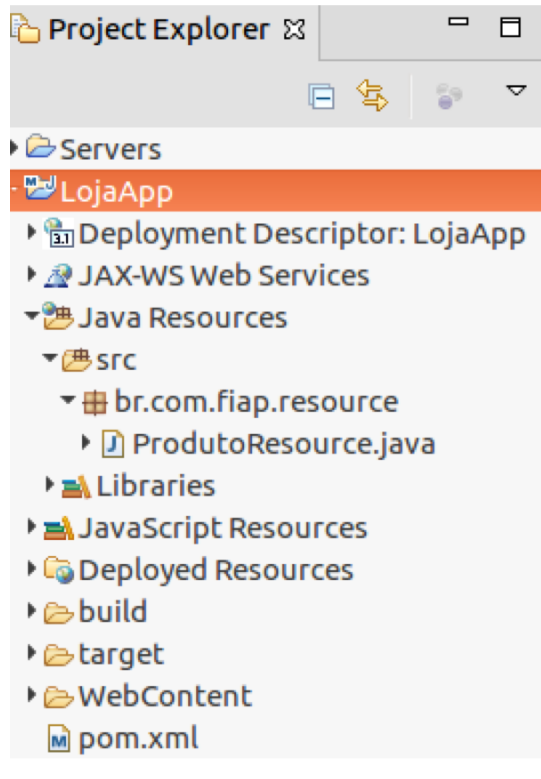
```
  <url-pattern>/rest/*</url-pattern>
```

```
</servlet-mapping>
```

Pacote onde estão as classes do web services

Parte da URL para acessar o web service

Vamos criar uma classe java no pacote configurado no web.xml (br.com.fiap.resource) chamada ProdutoResource, onde será desenvolvido os serviços.



```
import javax.ws.rs.GET;  
import javax.ws.rs.Path;  
import javax.ws.rs.Produces;  
import javax.ws.rs.core.MediaType;
```

```
@Path("/produto")
```

```
public class ProdutoResource {
```

```
    @GET
```

```
    @Produces(MediaType.TEXT_PLAIN)
```

```
    public String buscar(){
```

```
        return "Ola Mundo!";
```

```
    }
```

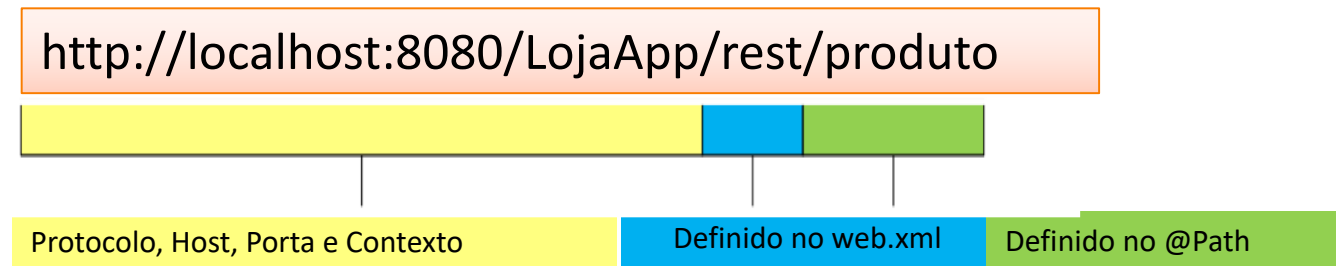
```
}
```

← Parte do endereço da URL para acessar o serviço:

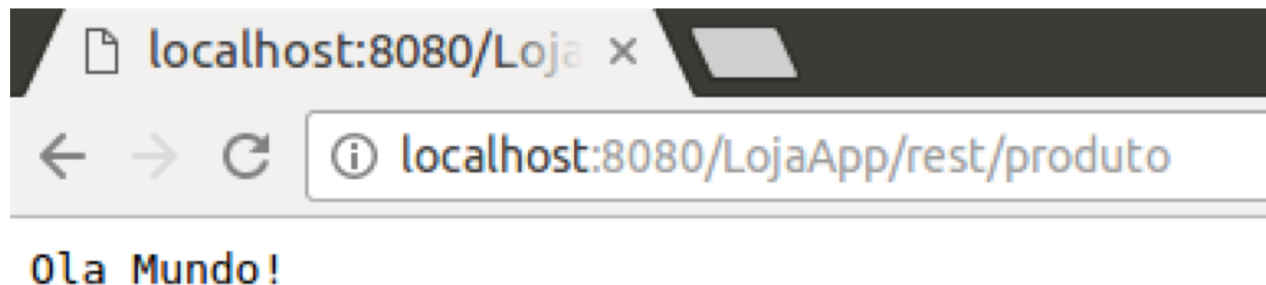
↑ Requisições GET e retorna um texto puro como resposta.

TESTE!

Execute o Servidor e faça uma chamada ao serviço através de sua **URL** no navegador:



Navegador:



Ao invés de trabalhar com o envio de Texto Puro, vamos trabalhar com **Json – JavaScript Object Notation**.

- Formato simples e leve para transferência de dados.
- Uma alternativa para o XML

Exemplo:

```
{  
  "show": "Oasis",  
  "preco": 150,  
  "local": "São Paulo"  
}
```

Exemplo de uma lista de Shows:

```
{  
  "shows": [  
    {  
      "show": "Oasis",  
      "preco": 150,  
      "local": "São Paulo"  
    },  
    {  
      "show": "Link Park",  
      "preco": 250,  
      "local": "Rio de Janeiro"  
    },  
    {  
      "show": "Jorge e Mateus",  
      "preco": 200,  
      "local": "São Paulo"  
    }  
  ]  
}
```

{ JSON }

Os colchetes [] limitam o array

EXEMPLO BUSCA (GET)

Vamos trabalhar com a biblioteca Jackson para converter objetos **Java** em representações **Json** e vice-versa.

A dependência já foi adicionada no projeto, dessa forma a biblioteca irá realizar a conversão automaticamente.

@XmlRootElement

public class ProdutoTO {

private int codigo;

private String titulo;

private double preco;

private int quantidade;

//construtores, gets e sets;

}

Vamos criar a classe para armazenar as informações do produto.

GET – LISTAR OS PRODUTOS

Vamos ajustar o código do serviço para que ele retorne todos os produtos cadastrados!

```
@Path("/produto")
public class ProdutoResource {

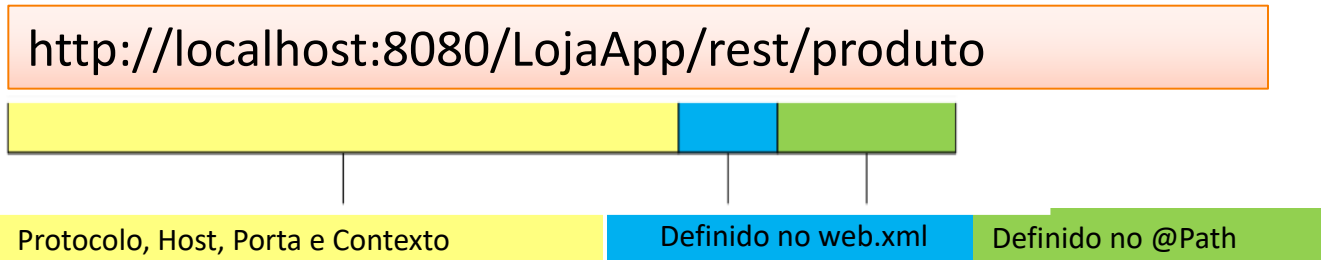
    private ProdutoBO produtoBo = new ProdutoBO();

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ProdutoTO> buscar(){
        return produtoBo.listar();
    }
}
```

← Tipo do retorno (JSON)

← Retorna a lista de produtos para ser convertido em um JSON array.

Execute o Servidor e faça uma chamada ao serviço através de sua **URL** no navegador:



GET – BUSCAR POR CÓDIGO

Agora vamos adicionar um serviço para recuperar um produto pelo seu código.

Parte da URL para acessar a busca com um parâmetro (id)

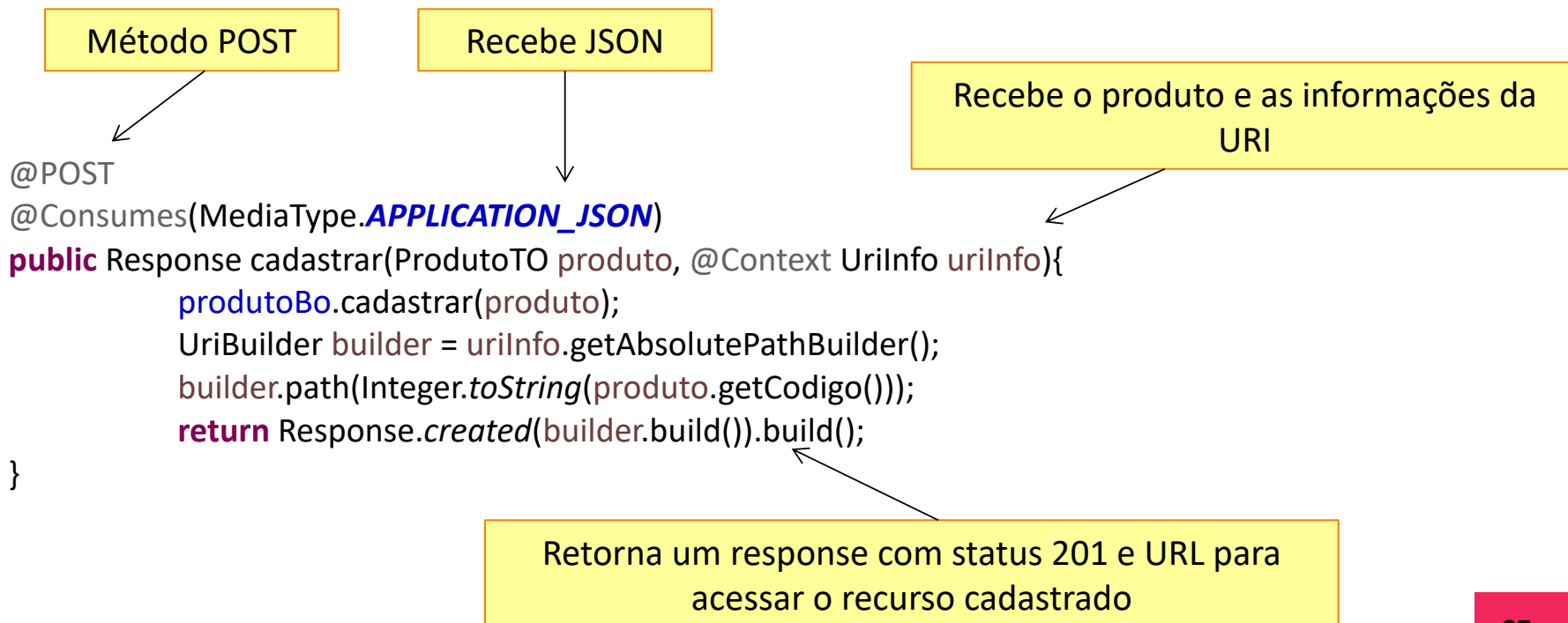
Tipo do retorno (JSON)

Injeta o parâmetro da URL no parâmetro do método

```
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public ProdutoTO buscar(@PathParam("id") int codigo){
    return produtoBo.buscar(codigo);
}
```

EXEMPLO CADASTRO (POST)

Agora vamos adicionar um método para cadastrar um Produto. O método recebe um objeto **ProdutoTO** e retorna um **Response**.



POST – TESTE!

Para enviar uma requisição POST, precisamos de um plugin no navegador, como **Postman**.

The screenshot displays the Postman interface for a POST request. The URL bar shows `http://localhost:8080/LojaApp/rest/produto/`, with an annotation "URL para o Resource" pointing to it. The request body is set to "raw" with the content type "JSON (application/json)", annotated with "Definição do Conteúdo da requisição.". The body content is a JSON object: `{ "titulo": "Xiaomi", "preco": 300, "quantidade": 20 }`, with an annotation "Conteúdo (Json)" pointing to it. The response section shows a status of "201 Created" and a time of "24 ms", annotated with "Resposta recebida.". The response headers include "Content-Length → 0", "Date → Fri, 10 Feb 2017 17:06:41 GMT", and "Location → http://localhost:8080/LojaApp/rest/produto/4".

http://localhost:8080/ x + No Environment

POST http://localhost:8080/LojaApp/rest/produto/ URL para o Resource

Params Send Save

Authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary JSON (application/json) Definição do Conteúdo da requisição.

```
1 {
2   "titulo": "Xiaomi",
3   "preco": 300,
4   "quantidade": 20
5 }
```

Conteúdo (Json)

Body Cookies Headers (3) Tests

Status: 201 Created Time: 24 ms

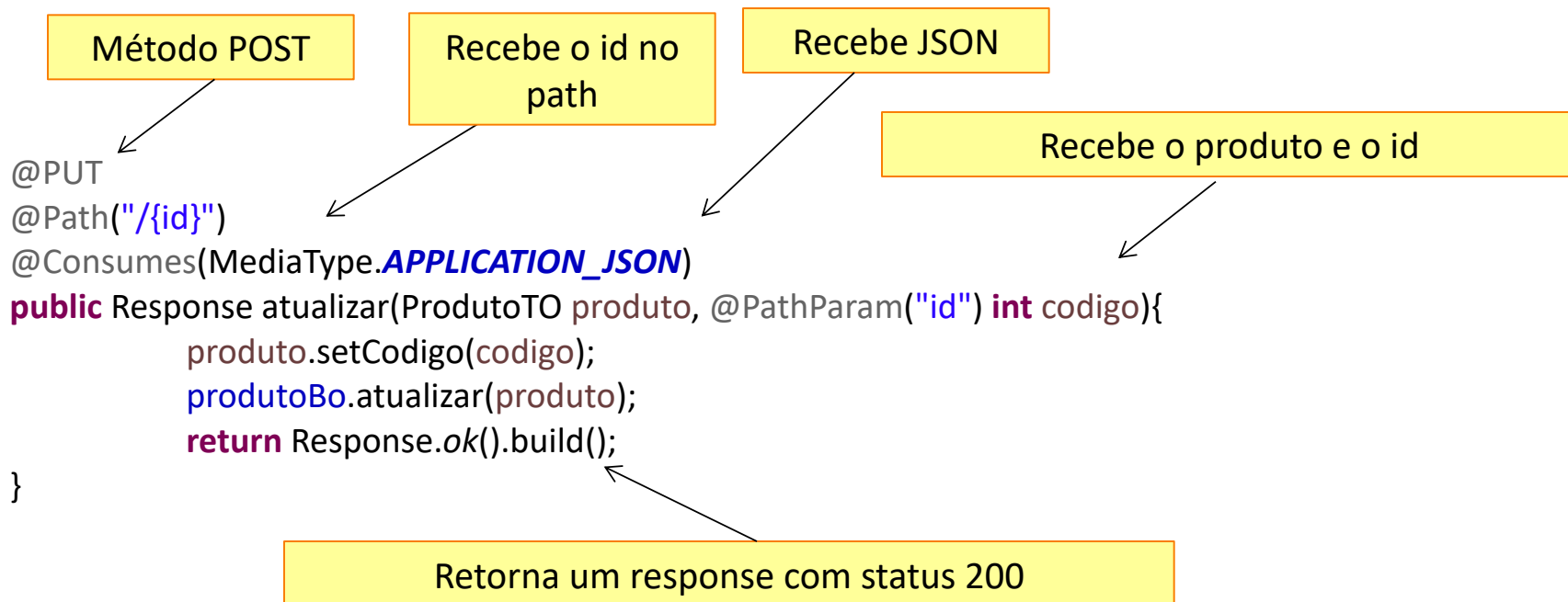
Content-Length → 0

Date → Fri, 10 Feb 2017 17:06:41 GMT

Location → http://localhost:8080/LojaApp/rest/produto/4

EXEMPLO ATUALIZAÇÃO (PUT)

O método para atualização recebe um objeto **ProdutoTO** e o **id** do objeto que será atualizado e retorna um **Response**.



PUT – TESTE!

Para enviar uma requisição **PUT** vamos utilizar o **Postman**.

The screenshot shows the Postman interface for a PUT request. The URL bar at the top displays `http://localhost:8080/`. Below it, the request method is set to **PUT** and the full URL is `http://localhost:8080/LojaApp/rest/produto/1`. The **Body** tab is selected, showing a JSON payload: `{ "titulo": "Xiaomi", "preco": 300, "quantidade": 20 }`. The response status is **200 OK** with a time of **17 ms**.

Annotations in the image:

- URL para o Resource com o id**: Points to the URL `http://localhost:8080/LojaApp/rest/produto/1`.
- Definição do Conteúdo da requisição.**: Points to the JSON body.
- Conteúdo (Json)**: Points to the JSON payload.
- Resposta recebida.**: Points to the **200 OK** status.

EXEMPLO EXCLUSÃO (DELETE)

O método de remoção recebe um código e não retorna nada (código HTTP 204)

Método DELETE

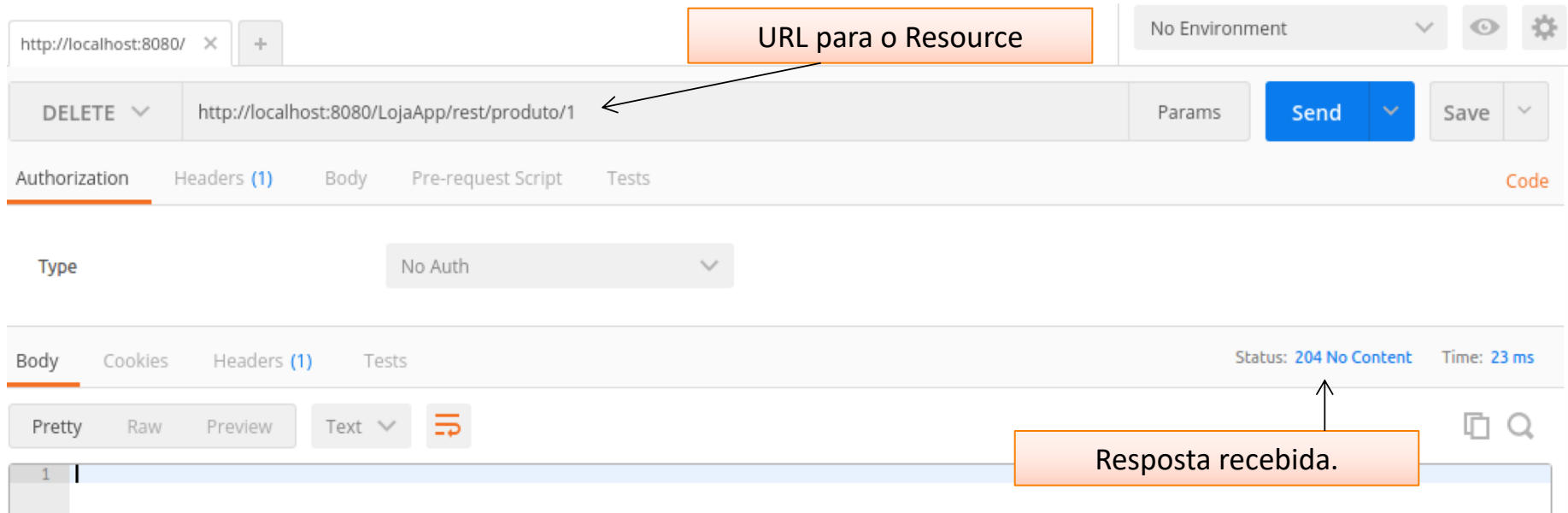
Parâmetro na URL

Recebe o id

```
@DELETE
@Path("/{id}")
public void remover(@PathParam("id") int codigo){
    produtoBo.remover(codigo);
}
```

POST – TESTE!

Para enviar uma requisição POST, precisamos de um plugin no navegador, como **Postman**.



Copyright © 2013 - 2018 Prof. Me. Thiago T. I. Yamamoto

Todos direitos reservados. Reprodução ou divulgação total ou parcial deste documento é expressamente proibido sem o consentimento formal, por escrito, do Professor (autor).

*"Em nossas vidas, a mudança é inevitável. A perda é inevitável.
A felicidade reside na nossa adaptabilidade em sobreviver a tudo de ruim" – Buda*