



# FIAP

## GRADUAÇÃO

45697056



TDS

# Responsive Web Development

Prof. Alexandre Carlos [profalexandre.jesus@fiap.com.br](mailto:profalexandre.jesus@fiap.com.br)

Prof. Luís Carlos [lsilva@fiap.com.br](mailto:lsilva@fiap.com.br)





# AGENDA

45697056



- Web Services Restful
  - URI, Métodos e Status Code do HTTP
- Web Services Restful com Java
  - JAX-RS, principais anotações
  - Criação do Projeto e configuração
  - Implementação do CRUD com JSON
- Tema 1
- Tema 1





# WEB SERVICE RESTFUL

45697056



## RESTFul (Representational State Transfer)

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





# MÉTODOS DO HTTP

45697056



O protocolo **HTTP** possui vários métodos, os principais:

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





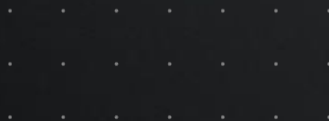
# URI – UNIFIED RESOURCE IDENTIFIER

45697056



Quando realizamos uma requisição, é preciso determinar o **endereço** do **recurso** que vamos acessar:

VERBO	URI	AÇÃO
POST	/pedido/	Criar
GET	/pedido/1	Visualizar
PUT	/pedido/1	Alterar
DELETE	/pedido/1	Apagar





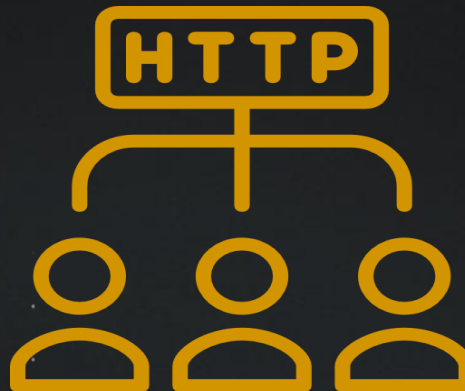
# HTTP STATUS CODE

45697056



Os códigos de **status das respostas HTTP** indicam se uma requisição HTTP foi corretamente concluída. As respostas são agrupadas em cinco classes: **respostas de informação, respostas de sucesso, redirecionamentos, erros do cliente e erros do servidor**;

- **1xx** – Informativa;
- **2xx** – Sucesso;
- **3xx** – Redirecionamentos;
- **4xx** – Erros do cliente;
- **5xx** – Erros do servidor;





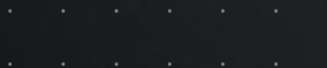
# HTTP STATUS CODE

45697056



Vamos trabalhar com alguns códigos na implementação do **Web Service**:

CODE	DESCRIÇÃO
200	Ok
201	Created (Criado)
204	No Content (Sem conteúdo)
500	Internal Server Error
404	Not Found
405	Method not Allowed







# HTTP STATUS CODE

---

45697056



**WEB SERVICE  
RESTFUL COM JAVA**





# WEB SERVICES RESTFUL - JAVA

45697056



## ■ JAX-RS

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





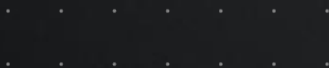
# JAX-RS - ANNOTATIONS

45697056



## ■ Principais anotações:

Anotação	Descrição
@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**

Create a standalone Java-based Web Application 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

EAR project name:

Working sets

☐ Add project to working sets

Working sets:

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

Module Version 4.0

Next para gerar o web.xml



# CRIANDO O PROJETO

New Dynamic Web Project

**Web Module**  
Configure web module settings.

Context root:

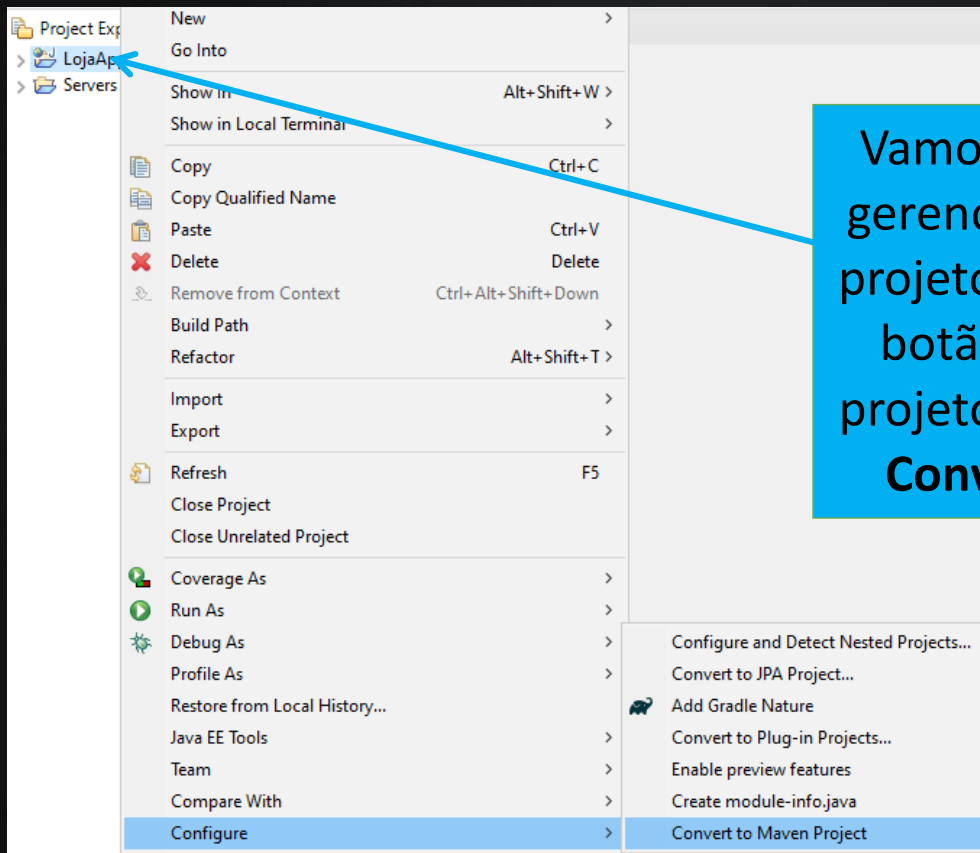
Content directory:

☒ Generate web.xml deployment descriptor

Marque para gerar o **web.xml**



# MAVEN



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**



# WEB SERVICES RESTFUL - JAVA

45697056



- **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.



**Apache Maven Project**

<http://maven.apache.org/>

<https://maven.apache.org/>



# DEPENDÊNCIAS POM.XML

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

Adicione as **dependências** após a tag **</build>**

```
<dependencies>
  <dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-server</artifactId>
    <version>2.35</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-servlet-core</artifactId>
    <version>2.35</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>2.25.1</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.inject</groupId>
    <artifactId>jersey-hk2</artifactId>
    <version>2.35</version>
  </dependency>
  <dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.1</version>
  </dependency>
</dependencies>
```





# DEPENDÊNCIAS MAVEN

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.

45697056





# CONFIGURAÇÃO RESTFUL

45697056



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

Pacote onde estão as classes  
do web services

Parte da URL para acessar o  
web service

```
<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>
```

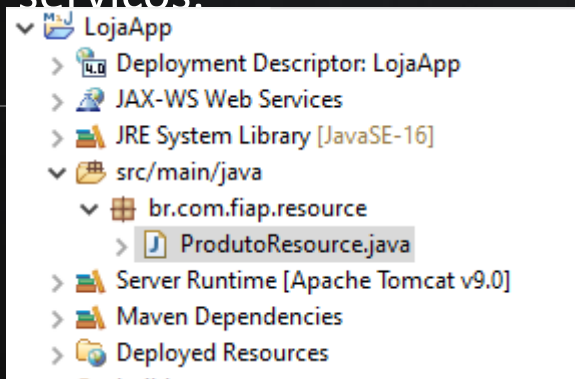


# CODAR!

45697056



- Vamos criar uma classe Java no pacote configurado no web.xml `br.com.fiap.resource` chamada `ProdutoResource`, onde será desenvolvido os serviços.



```
package br.com.fiap.resource;

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!

45697056



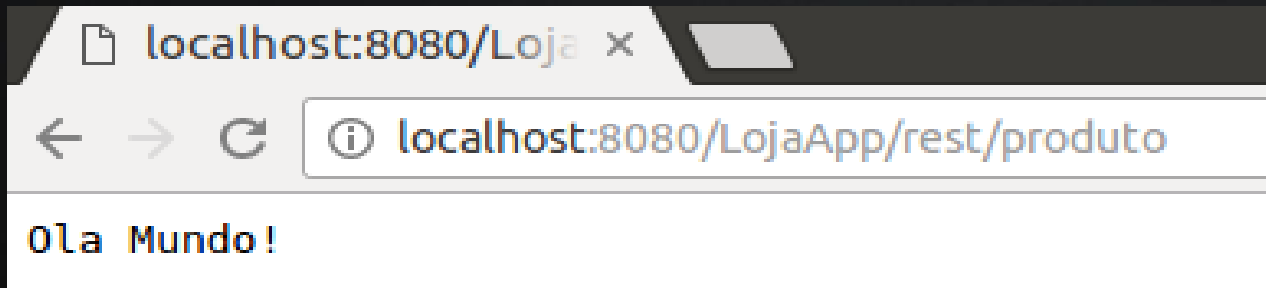
- Execute o Servidor e faça uma chamada ao serviço através de sua URL no navegador:

`http://localhost:8080/LojaApp/rest/produto`

Protocolo, Host, Porta e Contexto

Definido no `web.xml`

Definido no `@Path`





# EXERCÍCIO

45697056



1. Crie um projeto Web Service Restful. Vide o material para a configuração.
2. Crie uma Classe de serviço.
3. Crie a anotação na classe que identifique o path a ser acessado.
4. Nessa classe crie um método que retorne o saldo de uma conta corrente.
5. Anote o método com o tipo de dado de Aplicação TextPlain.
6. Anote o método de forma que ele seja acessado via **GET**.
7. Acesse via browser através da URL.





# JSON - ARRAY

Exemplo de um  
array em JSON:

Os colchetes [ ]  
limitam o array.

```
45697056
{
  "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 }

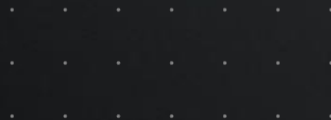


# EXERCÍCIO

45697056



1. Crie um projeto Web Service Restfull. Vide o material para a configuração.
2. Crie uma Classe de serviço.
3. Crie a anotação na classe que identifique o path a ser acessado.
4. Crie um método que retorne uma “[ ][ ]” matriz com as seguintes informações. No mínimo com 5 posições. E que possua os seguintes dados:
  - a) Nome;
  - b) CPF;
  - c) Email
5. Anote o método de forma que ele seja acessado via GET.
6. Anote o método com o tipo de dado de Aplicação APPLICATION\_JSON.
7. Acesse via browser através da URL do SERVIDOR.





# CRUD

---

45697056



## GET - BUSCAS







# JSON e JAVA

- 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**.

```
package br.com.fiap.resource;  
  
import javax.xml.bind.annotation.XmlRootElement;  
  
@XmlRootElement  
public class ProdutoTO {  
  
    private int codigo;  
    private String titulo;  
    private double preco;  
    private int quantidade;  
  
    //construtores, gets e sets;  
  
}
```

Anotação para identificar que a classe pode ser transformada em Json;

Crie a classe para **armazenar** as informações do **produto**.



# Classe DAO do projeto

- Vamos simular uma classe DAO no pacote `br.com.fiap.dao` que estaria acessando uma tabela no banco chamada `ProdutoDAO`. Nela vamos criar um `ArrayList` estático para usarmos como se fosse o banco. Também um método para buscar suas informações.

```
public class ProdutoDAO {  
  
    public static List<ProdutoTO> produto;  
  
    public ProdutoDAO() {  
  
        if(produto == null) {  
            produto = new ArrayList<ProdutoTO>();  
  
            ProdutoTO pto = new ProdutoTO();  
            pto.setCodigo(1);  
            pto.setPreco(27.99);  
            pto.setQuantidade(10);  
            pto.setTitulo("Grampeador");  
            produto.add(pto);  
  
            // mais 4 produtos  
        }  
    }  
}
```

```
public List<ProdutoTO> select(){  
    return produto;  
}
```





# Classe BO do projeto

- Agora vamos simular uma classe BO no pacote **br.com.fiap.bo** que usará os métodos da classe DAO chamada **ProdutoBO**.

```
public class ProdutoBO {  
  
    private ProdutoDAO pd;  
  
    public List<ProdutoTO> listar(){  
        pd = new ProdutoDAO();  
        return pd.select();  
    }  
  
}
```





# Classe ProdutoResource

- Vamos ajustar o código do nosso **ProdutoResource** para que ele retorne **todos** os **produtos** cadastrados ou somente **1**!

```
@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)





# GET - TESTE!

45697056



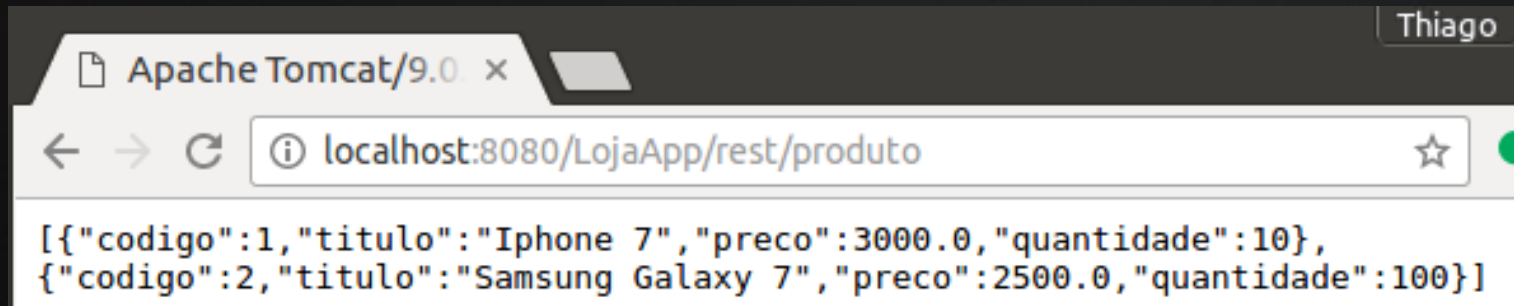
- Execute o Servidor e faça uma chamada ao serviço através de sua URL no navegador:

<http://localhost:8080/LojaApp/rest/produto>

Protocolo, Host, Porta e Contexto

Definido no **web.xml**

Definido no **@Path**





# Classe DAO do projeto

- Voltando para nossa classe **ProdutoDAO**, vamos criar um método para buscar apenas um produto.

```
public ProdutoTO select(int id){  
  
    for(int i = 0; i < produto.size(); i++) {  
        if(produto.get(i).getCodigo() == id) {  
            return produto.get(i);  
        }  
    }  
    return null;  
}
```





# Classe BO do projeto

- Agora vamos tratar esse método **ProdutoBO**.

```
public class ProdutoBO {  
  
    private ProdutoDAO pd;  
  
    public List<ProdutoTO> listar(){  
        pd = new ProdutoDAO();  
        return pd.select();  
    }  
  
    public ProdutoTO listar(int id){  
        pd = new ProdutoDAO();  
        return pd.select(id);  
    }  
}
```



# Classe ProdutoResource

- Vamos ajustar o código do nosso **ProdutoResource** para que ele retorne **apenas 1 produto** cadastrado!

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

    private ProdutoBO produtoBO = new ProdutoBO();

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

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

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

Annotacion que pega o id do  
path e passa como parâmetro





# GET - TESTE!

45697056



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

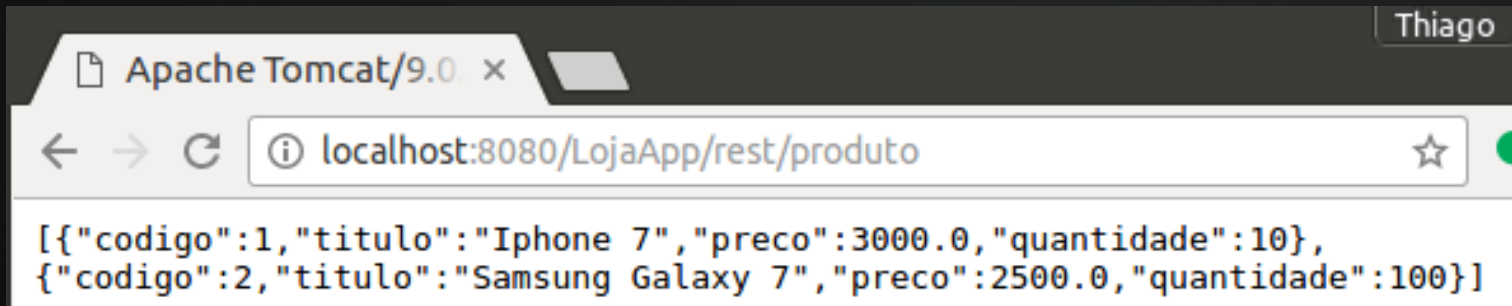
`http://localhost:8080/LojaApp/rest/produto/1`

Definido no `@PathParam`

Protocolo, Host, Porta e Contexto

Definido no `web.xml`

Definido no `@Path`





# EXERCÍCIO

45697056

1. Crie um projeto Web Service Restfull. Vide o material para a configuração.
2. Crie uma Classe de serviço.
3. Crie a anotação na classe que identifique o path a ser acessado.
4. Crie uma classe de transporte TO, adicione os seguintes atributos:
  - a) código
  - b) título
  - c) preço
  - d) distância
5. Crie uma classe de negócios que realize o intercâmbio de informações entre o resource e a base de dados (BO).
6. Crie uma classe para manipular os comandos com a camada de dados (DAO). Aqui vamos criar um banco de dados virtual estatico (`List<> = ArrayList<>`), para podermos acessar.
7. Popule esse banco virtual com pelo menos 10(dez) objetos TO.
8. Crie um método que retorne uma lista de objetos TO:
9. Anote o método de forma que ele seja acessado via GET.
10. Anote o método com o tipo de dado de `APPLICATION_JSON`.
11. Acesse via browser através da URL do SERVIDOR.



# CRUD

---

45697056



## POST - CADASTRO





# Classe DAO do projeto

- Vamos inserir novos cadastros no projeto. No **ProdutoDAO**, vamos criar um método insert.

```
public boolean insert(ProdutoTO pto) {  
    pto.setCodigo(produto.size() + 1);  
    return produto.add(pto);  
}
```





# Classe BO do projeto

- Agora vamos tratar esse método no **ProdutoBO**.

```
public boolean cadastrar(ProdutoTO pto) {  
    pd = new ProdutoDAO();  
    return pd.insert(pto);  
}
```





# Classe ProdutoResource

- Crie um método para **cadastar** um Produto. O método recebe um objeto **ProdutoTO** e retorna um **Response**

Método **POST**

Recebe **JSON**

Recebe o produto e as informações da **URI**

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response cadastrar(ProdutoTO produto, @Context UriInfo uriInfo) {
    produtoBO.cadastrar(produto);
    UriBuilder builder = uriInfo.getAbsolutePathBuilder();
    builder.path(Integer.toString(produto.getCodigo()));
    return Response.created(builder.build()).build();
}
```

Retorna um **response** com **status 201** e **URL** para acessar o recurso cadastrado





# POST - TESTE!

- Para enviar uma requisição **POST**, é preciso de uma ferramenta, como **Postman**.

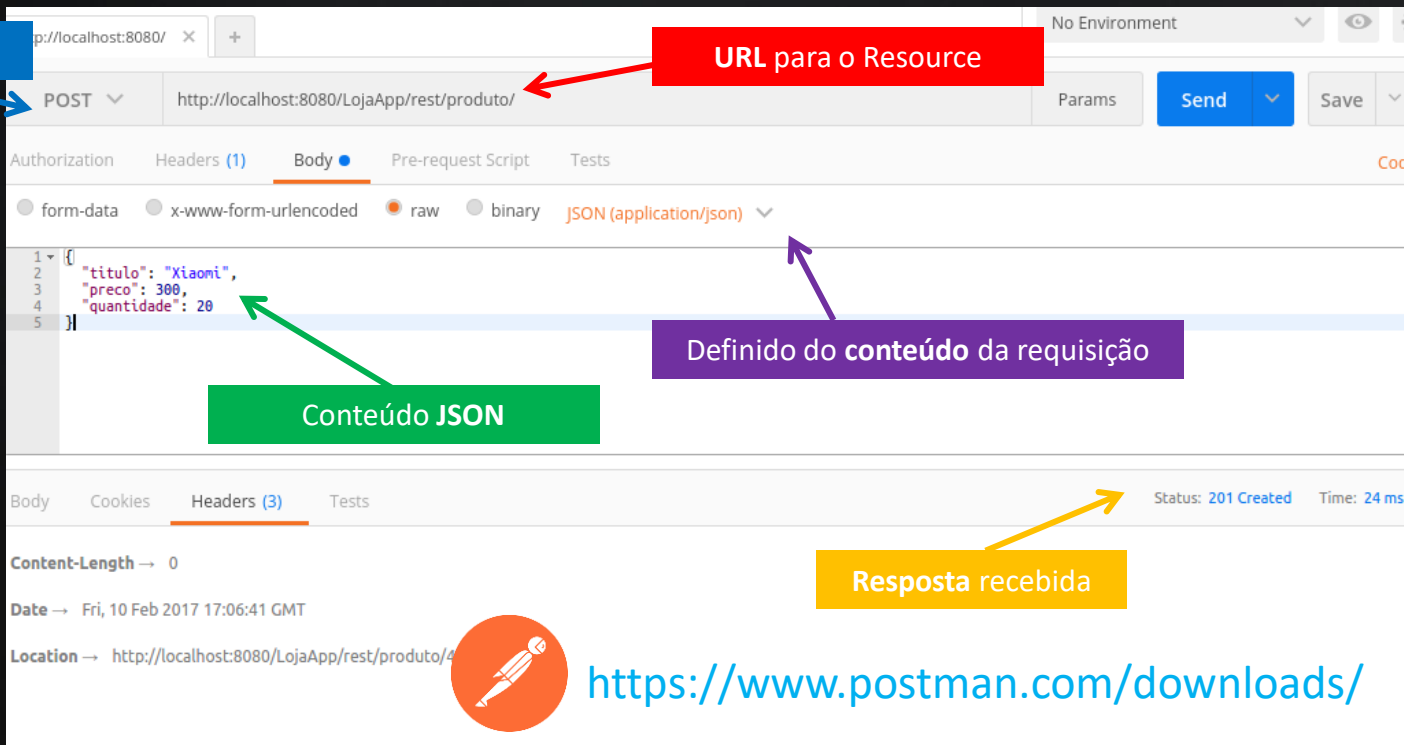
Método **POST**

URL para o Resource

Definido do conteúdo da requisição

Conteúdo JSON

Resposta recebida



<https://www.postman.com/downloads/>



# CRUD

---

45697056



## PUT - ATUALIZAÇÃO







# Classe DAO do projeto

- Vamos alterar cadastros já existentes no projeto. No **ProdutoDAO**, vamos criar um método update.

```
public void update(ProdutoTO pto) {  
  
    ProdutoTO p = select(pto.getCodigo());  
    p.setPreco(pto.getPreco());  
    p.setQuantidade(pto.getQuantidade());  
    p.setTitulo(pto.getTitulo());  
  
}
```





# Classe BO do projeto

- Agora vamos tratar esse método no **ProdutoBO**.

```
public void atualiza(ProdutoTO pto) {  
    pd = new ProdutoDAO();  
    pd.update(pto);  
}
```





# Classe ProdutoResource

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

Método PUT

Recebe o JSON

Recebe o id no  
path (URL)

```
@PUT
@Path("/{id}")
@Consumes(MediaType.APPLICATION_JSON)
public Response atualiza(ProdutoTO produto, @PathParam("id") int id)
{
    produto.setCodigo(id);
    produtoBO.atualiza(produto);

    return Response.ok().build();
}
```

Recebe o produto e o id

Retorna um response com HTTP status 200





# PUT - TESTE!

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

**Método PUT**

**URL para o Resource com o id**

**Definido do conteúdo da requisição**

**Conteúdo JSON**

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

**Resposta recebida**

Status: 200 OK Time: 17 ms



# CRUD

---

45697056



## DELETE - REMOÇÃO





# Classe DAO do projeto

- Agora só falta a remoção de registros do cadastros no projeto. No **ProdutoDAO**, vamos criar um método remove.

```
public void delete(int id) {  
    produto.remove(select(id));  
}
```





# Classe BO do projeto

- Agora vamos tratar esse método no **ProdutoBO**.

```
public void remover(int id) {  
    pd = new ProdutoDAO();  
    pd.delete(id);  
}
```





# Classe ProdutoResource

- O método de remoção **recebe** um **código** e não retorna nada (código HTTP 204 (no content));

Método DELETE

Recebe o id no  
path (URL)

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

Recebe o id







# DELETE - TESTE!

- Utilize o **postman** para testar a função de **remoção**.

Método **DELETE**

URL para o Resource com o id

The screenshot shows the Postman interface with a DELETE request configured. The URL is `http://localhost:8080/LojaApp/rest/produto/1`. The response status is `204 No Content` and the time taken is `23 ms`. The response body is empty.

Annotations:

- A blue arrow points from the **Método DELETE** text to the **DELETE** dropdown menu.
- A red arrow points from the **URL para o Resource com o id** text to the URL field.
- A yellow arrow points from the **Resposta recebida** text to the `204 No Content` status.



# Conflito com o CORS do projeto

- O CORS é um mecanismo que os navegadores utilizam para compartilhar recursos entre diferentes origens. Ele é uma especificação da W3C e utiliza o Header do HTTP para informar aos navegadores se determinado recurso pode ou não ser acessado.
- Como nossa API será consumida por um SPA, vamos criar a autorização de uso. Crie uma classe chamada CorsFilter no pacote resource.

```
package br.com.fiap.resource;

import java.io.IOException;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.ext.Provider;

@Provider
public class CorsFilter implements ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext,
        ContainerResponseContext responseContext) throws IOException {
        responseContext.getHeaders().add(
            "Access-Control-Allow-Origin", "*");
        responseContext.getHeaders().add(
            "Access-Control-Allow-Credentials", "true");
        responseContext.getHeaders().add(
            "Access-Control-Allow-Headers",
            "origin, content-type, accept, authorization");
        responseContext.getHeaders().add(
            "Access-Control-Allow-Methods",
            "GET, POST, PUT, DELETE, OPTIONS, HEAD");
    }
}
```



# EXERCÍCIO

45697056



Vamos dar continuidade no projeto do exercício anterior, implemente nas classes DAO, BO e RESOURCE os métodos necessários para criar as ações de cadastrar, alterar e remover transportes no projeto.



# DUVIDAS





Copyright © 2015 - 2021 Prof. Luís Carlos S. Silva  
Prof. Alexandre Carlos de Jesus

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