Sistemas Web Seguros

UNIDADE 04

Serviços web REST

Nesta unidade, conheceremos características importantes do *Hypertext Transfer Protocol* (HTTP) para desenvolvimento de serviços *web*, os fundamentos do protocolo de comunicação REST e como a especificação JAX-RS nos ajudará a construir *web services* na linguagem Java. Construiremos exemplos e evoluiremos o sistema SISRH para, além dos serviços SOAP, ofertar serviços REST.

HTTP

Antes dos serviços REST, precisamos conhecer alguns aspectos do HTTP que talvez você não tenha percebido. Nos primórdios da internet, o HTTP ganhou fama por ser o principal protocolo para acessar páginas HTML, mas, além de fornecer páginas, ele é

utilizado para executar ações por aplicativos na *web*. Praticamente tudo que você faz na rede, como postar uma foto, curtir um *feed* e comentar no seu grupo familiar, passa pelo HTTP.

Todas essas e outras ações geram pedidos aos servidores na nuvem, chamados requisições (*request*) HTTP.

Veja alguns exemplos de requisições – destacamos a supressão do sufixo do domínio (http://dominio.com):

- Para enviar uma mensagem*:* `POST /mensagem`
- Para ligar uma lâmpada*:* `POST /lampada`
- Para enviar uma reclamação*:* `POST /reclamacao`
- Para gerar um alerta*: : POST /alerta*

Perceba que as requisições HTTP fazem referência a um **recurso**: "mensagem", "lâmpada", "reclamação" e "alerta", mas até o momento parece não fazer muito sentido, certo?

Onde está o conteúdo da mensagem? Ou o comando para ligar a lâmpada? Ou os dados da reclamação? Ou a mensagem de alerta?

As informações dos **recursos** são enviadas por meio do ***payload*** da requisição HTTP. O *payload* ("carga", em português) representa o conteúdo enviado por um meio de transporte, como, no nosso caso, requisições HTTP.

Podemos construir requisições HTTP para certo **recurso** com seu respectivo ***payload*** utilizando o comando **cURL** – cliente HTTP existente nos sistemas Unix e Windows.

Para enviar uma mensagem: POST /mensagem.

```
curl -X POST -d "Olá, você está na disciplina Sistemas Web Seguros" http://dominio.com/mensagem
```

Para ligar uma lâmpada: POST /lampada.

```
curl -X POST -d "{'comando':'ligar'}" http://dominio.com/lampada
```

Para enviar uma reclamação: POST /reclamacao.

```
curl -X POST -d
"produto=geladeira&motivo=atraso&mensagem=Quero saber quando
vou receber o produto." http://dominio.com/reclamacao
```

Para gerar um alerta: POST /alerta.

```
curl -X POST -d "{'valor:'vermelho'}"
http://dominio.com/alerta
```

As operações ilustradas geram uma ação, por isso utilizamos o método **POST**. As informações geradas, como o alerta ou a reclamação, poderiam ser lidas por uma requisição HTTP utilizando o método **GET**.

```
curl -X GET http://dominio.com/mensagem/36598
curl -X GET http://dominio.com/lampada/65242
curl -X GET http://dominio.com/reclamacao/8eu387
curl -X GET http://dominio.com/alerta/69898
```

Perceba que os recursos são lidos a partir de um identificador; nos exemplos: a mensagem 36598; a lâmpada 65242; a reclamação 8eu387; e o alerta 69898.

Se quiséssemos remover um recurso, poderíamos enviar uma requisição HTTP por meio do método **DELETE**.

```
curl -X DELETE http://dominio.com/mensagem/36598
```

Para sobrescrever um recurso existente, por exemplo, em uma alteração, poderíamos usar o método **PUT**. Imagine que o alerta 69898 mudará da cor vermelha para a amarela:

```
curl -X PUT -d "{'valor:'amarelo'}"
http://dominio.com/alerta/69598
```

Você acabou de conhecer os quatro principais métodos HTTP utilizados na internet para **incluir**, **alterar**, **consultar** e **excluir** dados, respectivamente, por meio dos métodos **POST**, **PUT**, **GET** e **DELETE**.

Códigos de *status* HTTP

Quando um servidor *web* recebe uma requisição HTTP, que pode ser um GET, POST, PUT ou DELETE, ele tem de retornar uma resposta. As respostas são criadas e possuem um código de *status* padronizado e organizado por diversas faixas de valores:

- **Entre 100 e 102:** são retornos informativos e de pouca importância, que provavelmente nunca utilizará.
- **Entre 200 e 299:** são códigos para representar **sucesso**. Esta é uma faixa importante, pois temos várias situações de sucesso, tais como: 200 para um OK; 201 para informar que o recurso foi criado; e 202 para registrar um aceite.
- **Entre 300 e 399:** são códigos que representam **redirecionamento**; por exemplo, o 302 informa que o recurso de destino reside temporariamente em URL diferente.
- **Entre 400 e 499:** representam erros da requisição enviada pelo cliente. Por exemplo, uma requisição malformada terá como retorno 400, enquanto uma requisição sem autorização recebe 401 e uma requisição para um recurso que não existe tem como retorno 404.
- **Entre 500 e 599:** representam erros do servidor, como um erro interno (500), um erro de função não implementada (501), execução em *loop* (508) ou falha de armazenamento (507).

Você pode ter acesso aos principais códigos de situação HTTP no *site* http://httpstatus.es.

Cabeçalho

Os cabeçalhos (*headers*) servem para complementar as informações de uma requisição HTTP. Tenha cuidado para não confundir dados do *payload* com os dados do cabeçalho. O *payload* terá as informações relacionadas aos recursos do sistema, enquanto o cabeçalho terá informações auxiliares, como, por exemplo, se o conteúdo do *payload* está no formado XML ou JSON. Veja este exemplo:

```
curl -X POST -d "{'valor:'amarelo'}" -H "Content-Type:application/json" http://dominio.com/alerta/69598
```

Outro importante exemplo de uso do cabeçalho é o envio de credenciais de acesso, como usuário e senha ou *tokens* de acesso. Tais informações não são passadas no *payload*, pois devem ser registradas nos cabeçalhos.

Payload

O *payload* é a parte da requisição HTTP que possui os dados dos recursos, também podendo ser chamado *body*. Geralmente, é utilizado na inclusão (método POST) ou alteração de dados (método PUT) e menos empregado nas requisições de leitura (método GET) e exclusão (método DELETE).

O *payload* pode ser formado com os seguintes tipos de dado:

Texto

- *text/plain:* texto puro sem nenhum formato. Geralmente, uma mensagem, uma palavra, uma frase.
- *text/html:* texto no formato HTML.
- *text/csv:* texto com valores separados por vírgula.
- *text/vcard:* texto no formato vCard, que representa cartões de apresentação eletrônicos.

Imagem

- *image/png:* imagem no formato PNG.
- *image/gif:* imagem no formato GIF.
- *image/jpeg:* imagem no formato JPEG.

Vídeo

- *video/mp4:* vídeo no formato MP4.
- *video/avi:* vídeo no formato AVI.

Áudio

- *audio/mp4:* áudio no formato MP4.
- *audio/mpeg:* áudio no formato MPEG.
- *audio/vorbis: stream* de áudio no formato Vorbis.

Multipropósito

- *application/json:* dados no formato JSON.
- *application/xml:* dados no formato XML.
- *application/x-www-form-urlencoded:* dados no formato FORM chave=valor&chave=valor.
- *application/octet-stream:* arquivos binários diretos no *body.*
- *application/gzip:* dados compactados no padrão Gzip.

Usado para *upload*

- *multipart/form-data:* suporte a vários arquivos e atributos no mesmo *body.*

Especificação JAX-RS

Da mesma forma que conhecemos, na Unidade 2, a especificação JAX-WS para construção de serviços SOAP em Java, agora teremos a especificação Jakarta RESTful Web Services (JAX-RS), que nos fornece suporte para criação de serviços *web* REST. Lembra-se dos métodos HTTP GET, POST, PUT e DELETE para manipulação de recursos? Com o JAX-RS, teremos uma forma simplificada para tratar esses métodos na linguagem Java.

Vamos ao exemplo.

Imagine que, a partir da seguinte requisição, poderemos obter os dados do empregado 87:

```
curl -X GET http://dominio.com/empregado/87
```

Utilizando a especificação JAX-RS, teríamos a respectiva classe com o método **GET** para tratar o pedido de leitura ao empregado 87:

```
@Api
@Path("/empregado")
public class EmpregadoREST {

	@GET
	@Path("{id}")
	@Produces(MediaType.APPLICATION_JSON)
	public Response obterEmpregado(@PathParam("id") String id)
{
		Empregado emp = //obter do banco
		return Response.ok().entity(emp).build();
	}
}
```

Perceba que o JAX-RS utiliza outras anotações:

- `@Api`: indica que a classe possui uma API REST.
- `@Path("/empregado")`: indica o caminho da API.
- `@GET`: anota que o método **obterEmpregado** atenderá às requisições HTTP com o método GET.
- `@Path("{id}")`: indica a formação da chamada; no exemplo `/empregado/97`, o id receberá o valor 97.
- `@Produces`: informa que o serviço retornará com certo formato, neste caso, na estrutura JSON.

- `@PathParam("id")`: indica que o valor será atribuído à variável **id**.

Veja um exemplo possível para o retorno dessa chamada, que foi formatada em JSON:

```
{
   "id": "eb5a93a1-7977-4875-9344-89b23412599b",
   "nome": "Ricardo"

}
```

Agora, veremos um exemplo para tratar o método **POST**, utilizado geralmente para inserir registros:

```
@POST
	@Consumes(MediaType.APPLICATION_JSON)
	@Produces(MediaType.APPLICATION_JSON)
	public Response incluirEmpregado(Empregado emp) {
		//incluir empregado no banco
		return Response.ok(emp).build();
	}
```

Perceba que o método recebeu a anotação **@POST**, pois se trata de uma inclusão. O serviço receberá dados (`@Consumes`) no formato JSON e produzirá (`@Produces`) dados de resultado também no formato JSON.

Segundo o HTTP, o método para alteração de dados será feito pelo método **PUT**, representado pela anotação **@PUT**.

```
@PUT
	@Path("{id}")
	@Consumes(MediaType.APPLICATION_JSON)
	@Produces(MediaType.APPLICATION_JSON)
	public Response alterarEmpregado(@PathParam("id") String
id, Empregado emp) {
		//alterar no banco...
		return Response.ok().entity(emp).build();
	}
```

Finalmente, a remoção de dados é feita pelo método HTTP **DELETE**, representado pela anotação **@DELETE.**

```
@DELETE
	@Path("{id}")
	@Consumes(MediaType.APPLICATION_JSON)
	@Produces(MediaType.APPLICATION_JSON)
```

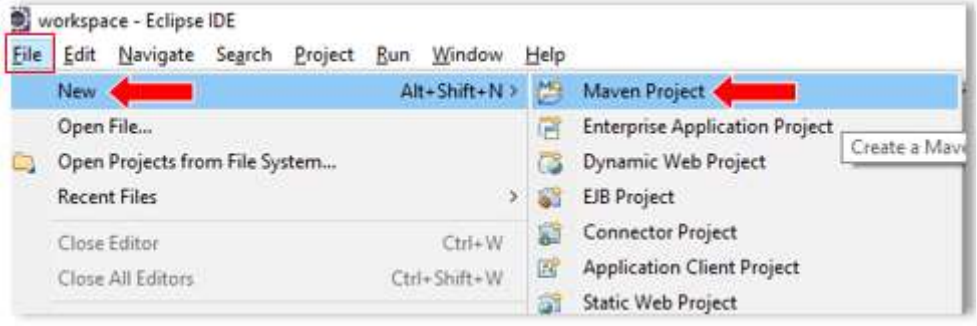
```
        public Response excluirEmpregado(@PathParam("id") String
id) {
                // excluir no banco...
                return Response.ok().build();
        }
```

Construindo um *web service* REST

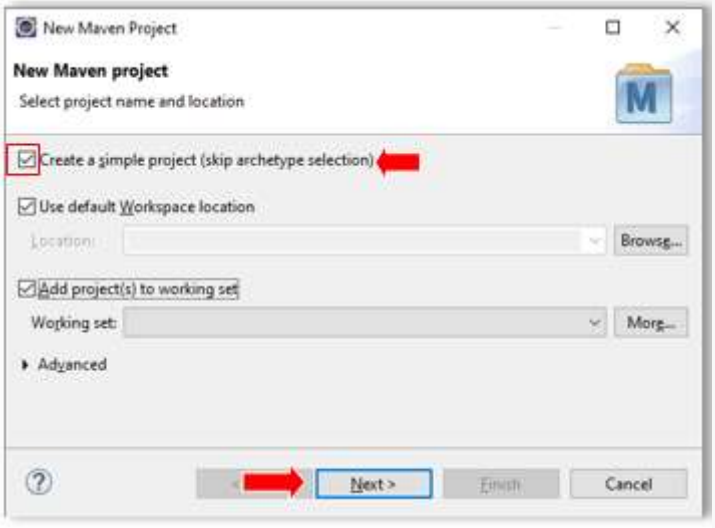
Para nos aprofundarmos nos conceitos estudados, vamos criar um exemplo simples de serviço REST utilizando a especificação JAX-RS. Nesse serviço, armazenaremos em memória objetos do tipo **empregado**.

Criação do projeto de exemplo: projetorest

Abra seu Eclipse IDE e crie um projeto Maven.



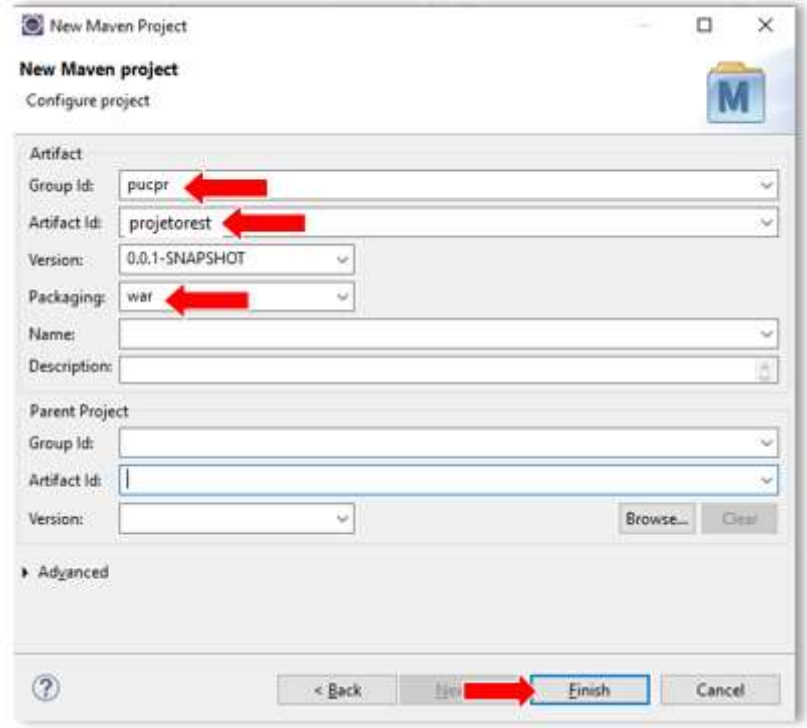
Escolha a opção **Create a simple project** para criar um projeto com uma estrutura básica.



Na tela de configuração do projeto, informe os campos:

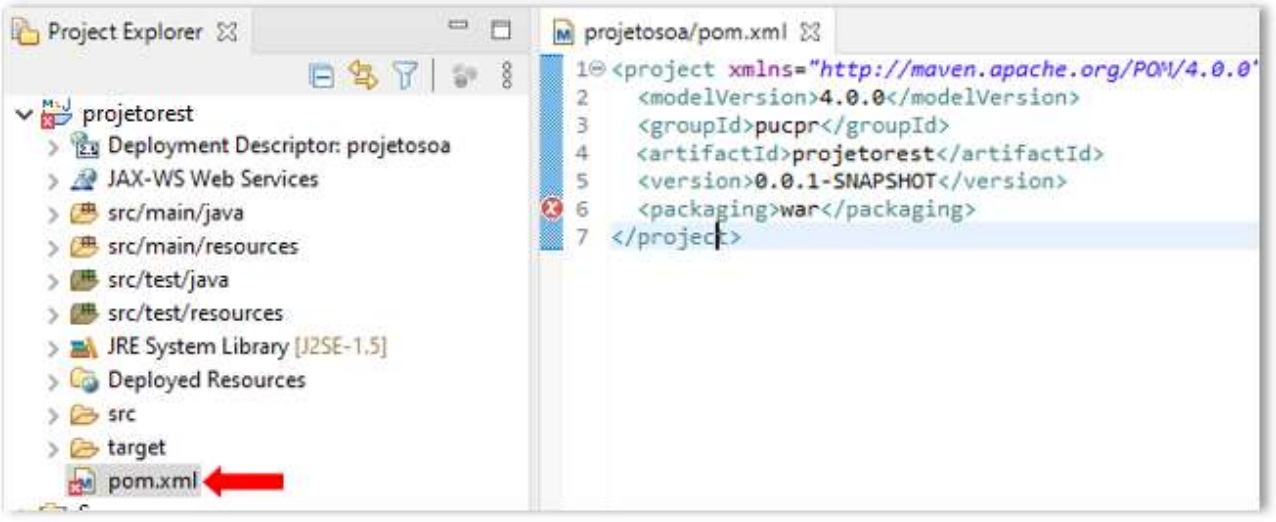
- Group Id: **pucpr**
- Artifact id: **projetorest**
- Packaging: **war**

Clique no botão **Finish** e aguarde a criação do projeto.



Aguarde até o projeto ser exibido, conforme imagem a seguir. Não se preocupe se houver alguma indicação de erro (>), vamos resolver isso configurando nosso projeto para que ele se transforme em uma solução *web service* REST.

Primeiramente, abra o arquivo **pom.xml**.



Logo abaixo da *tag* `</packaging>` , inclua a dependência para a biblioteca do projeto Jersey, que implementa a especificação JAX-RS.

```
<dependencies>
        <!-- jax-rs -->
      <dependency>
                <groupId>com.sun.xml.ws</groupId>
                <artifactId>jaxws-rt</artifactId>
                <version>2.3.2</version>
        </dependency>
        <dependency>
                <groupId>org.glassfish.jersey.core</groupId>
                <artifactId>jersey-server</artifactId>
                <version>2.6</version>
        </dependency>
        <dependency>
                <groupId>org.glassfish.jersey.containers</groupId>
                <artifactId>jersey-container-servlet</artifactId>
                <version>2.6</version>
        </dependency>
        <dependency>
                <groupId>com.fasterxml.jackson.jaxrs</groupId>
                <artifactId>jackson-jaxrs-json-
provider</artifactId>
                <version>2.4.1</version>
        </dependency>

        <!-- swagger -->
        <dependency>
                <groupId>io.swagger</groupId>
                <artifactId>swagger-jaxrs</artifactId>
                <version>1.5.7</version>
        </dependency>
</dependencies>
```

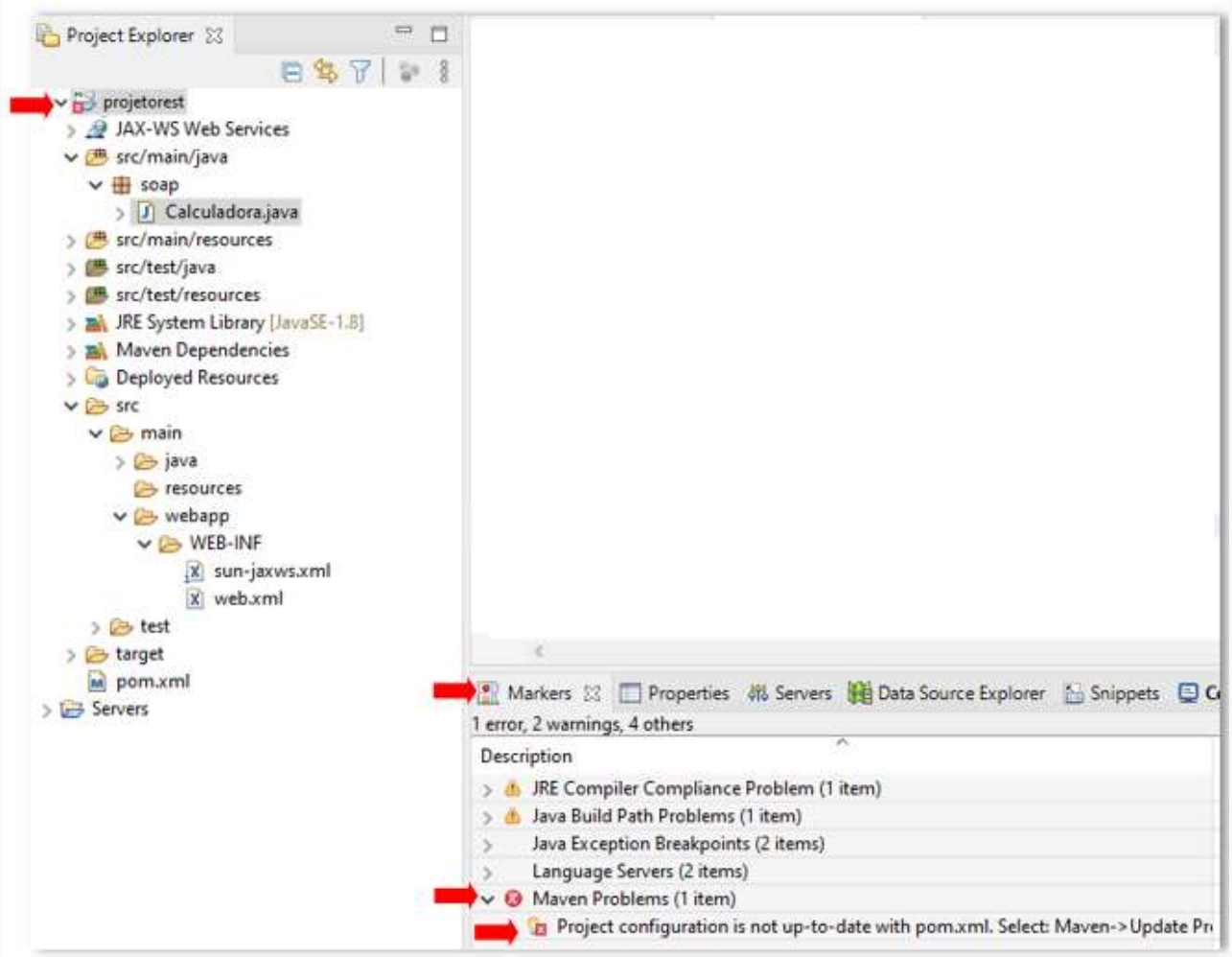
Em seguida, abaixo da *tag* *</dependencies>*, inclua a seguinte configuração de *build*, que indicará que nosso sistema será compatível com o Java a partir da versão 1.10 e que nosso projeto será identificado com o nome **projetores**:

```
<build>
        <pluginManagement>
```

```
                <plugins>
                        <plugin>

<groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-compiler-
plugin</artifactId>
                                <version>3.6.1</version>
                                <configuration>
                                        <source>1.10</source>
                                        <target>1.10</target>
                                </configuration>
                        </plugin>
                        <plugin>

<groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-war-
plugin</artifactId>
                                <version>3.0.0</version>
                                <configuration>

<warName>projetorest</warName>
                                </configuration>
                        </plugin>
                </plugins>
        </pluginManagement>
</build>
```

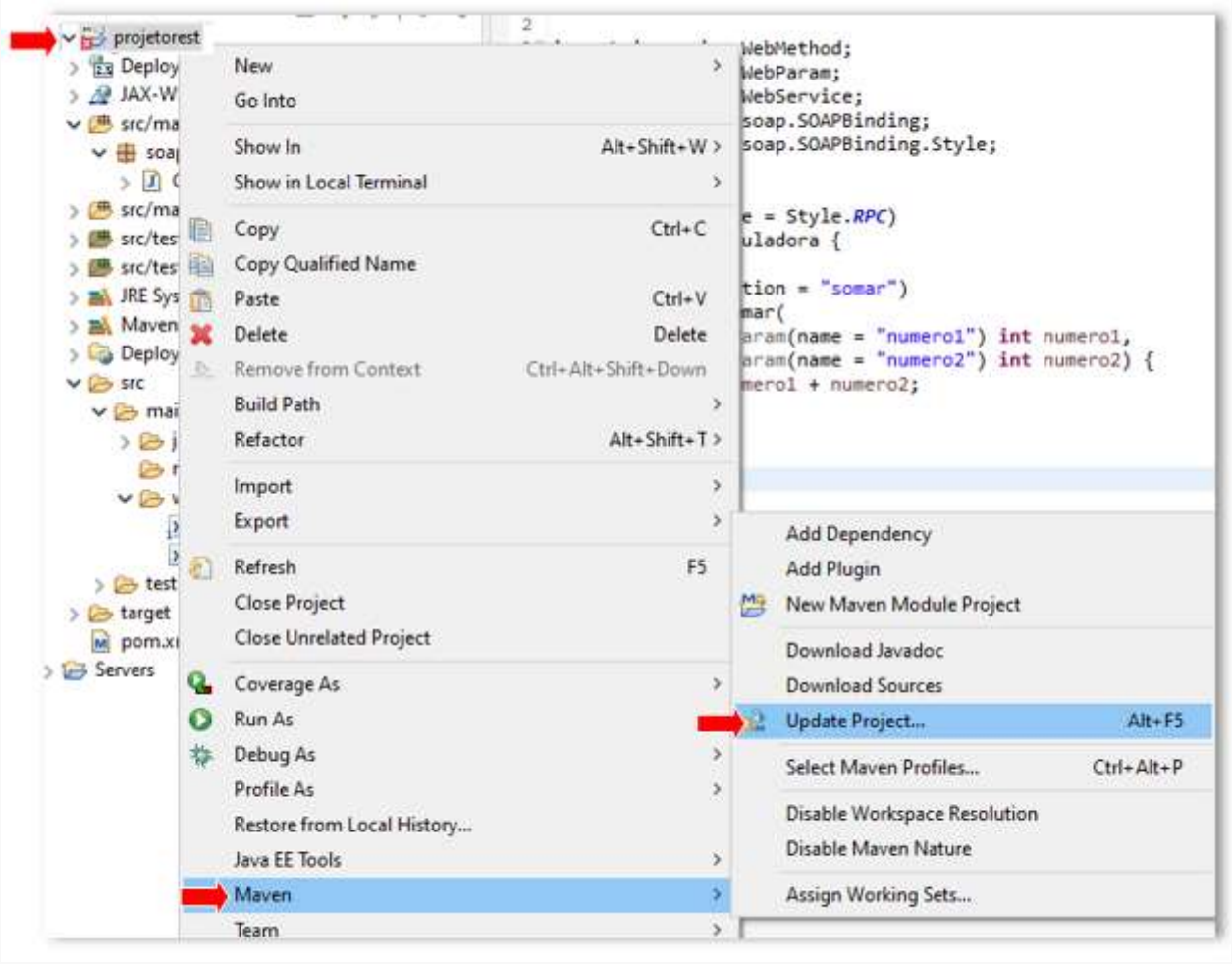
Nosso projeto está configurado!

Caso esteja aparecendo um ícone de erro, você poderá ver os detalhes sobre ele na aba **Markers**. Provavelmente, o erro se dá porque seu projeto Maven precisa ser atualizado. Isso geralmente acontece quando modificamos o arquivo **pom.xml** e não aplicamos a atualização.

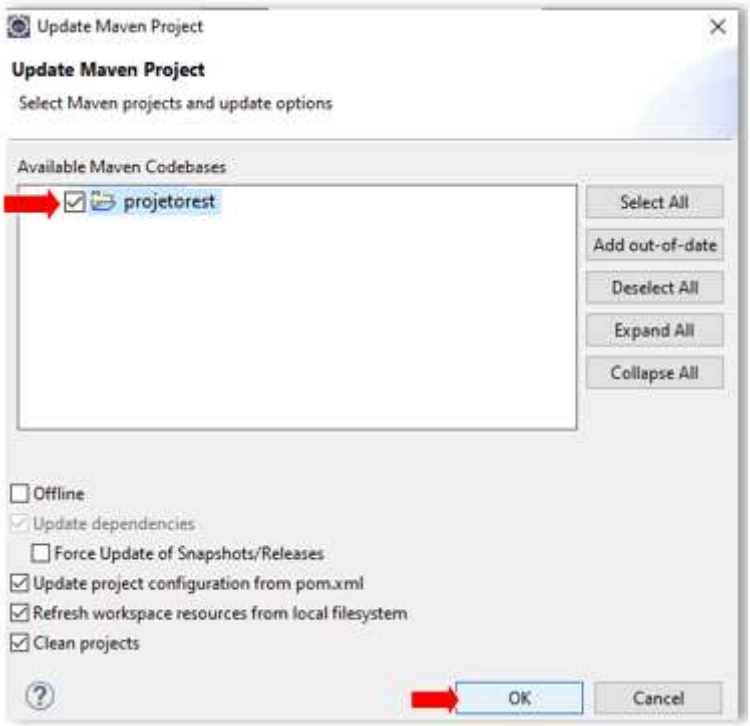


Esse erro indica que precisamos atualizar a leitura do arquivo **pom.xml**.

Clique com o botão direito do *mouse* no projeto e acione o menu **Maven**.



Deixe seu projeto marcado e clique em **OK**.



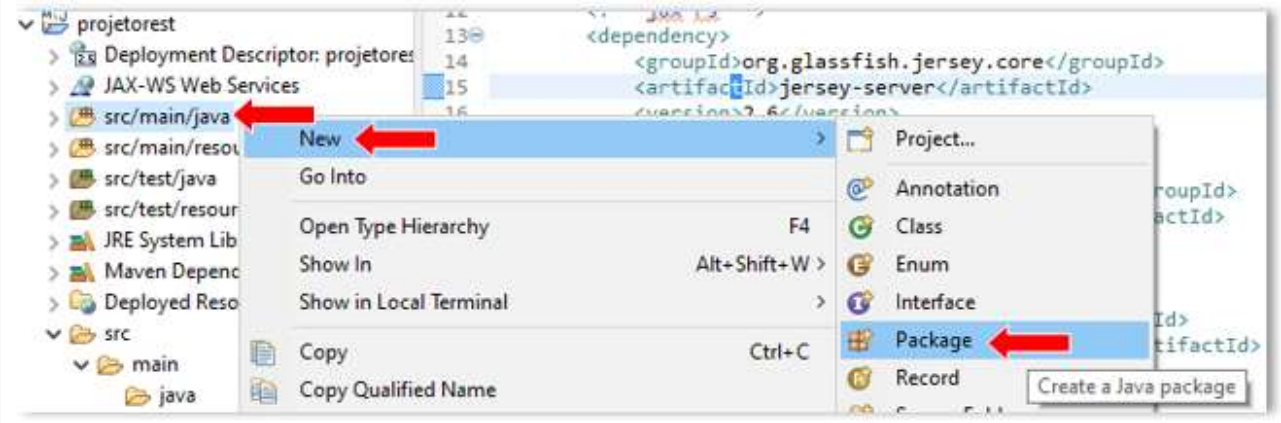
Isso resolverá a indicação de problema. Sempre que seu arquivo **pom.xml** for alterado, recomendamos fazer essa atualização. Ainda podem aparecer mensagens de aviso (*warnings*), mas não vamos nos preocupar com elas; nosso projeto está pronto para ser

executado.

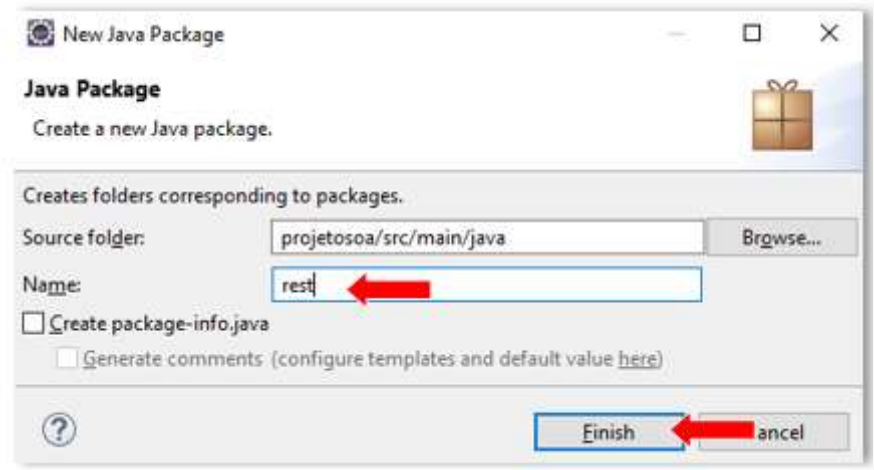
Para iniciar a construção do primeiro *web service* REST, vamos criar o seguinte padrão de URL:

`http://localhost:8080/projetorest/rest/`**`empregado`**

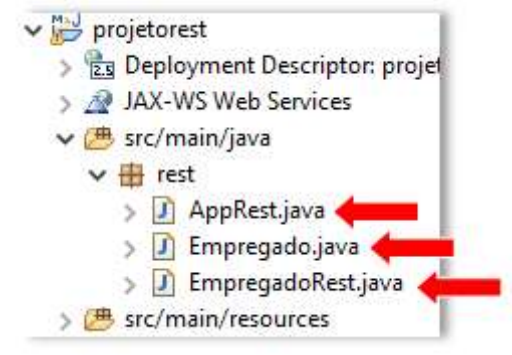
Para organizar nossos códigos, crie o pacote **rest**, clicando com o botão direito do *mouse* na pasta **src/main/java**.



Informe o nome do pacote: **rest**.



Crie três classes Java dentro do pacote **rest**: AppRest, Empregado e EmpregadoRest.



A classe **Empregado** será bem simples. Vamos guardar duas informações: um id e um nome.

```
package rest;

public class Empregado {

	private String id;
	private String nome;

	public Empregado() {
		super();
	}

	public Empregado(String id, String nome) {
		super();
		this.id = id;
		this.nome = nome;
	}

	public String getId() {
		return id;
	}

	public void setId(String id) {
		this.id = id;
	}

	public String getNome() {
		return nome;
	}
```

```
	public void setNome(String nome) {
		this.nome = nome;
	}

}
```

A classe **EmpregadoRest** possui as operações básicas de consulta: inclusão, alteração e exclusão.

```
package rest;

import java.util.*;
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import io.swagger.annotations.Api;

@Api
@Path("/empregado")
public class EmpregadoRest {

	static private HashMap<String, Empregado> empregados = new
HashMap<String, Empregado>();

	@GET
	@Path("{id}")
	@Produces(MediaType.APPLICATION_JSON)
	public Response obterEmpregado(@PathParam("id") String id)
{
		if (empregados.containsKey(id)) {
			return
Response.ok().entity(empregados.get(id)).build();
		} else {
			return Response.status(404).build();
		}
	}

	@GET
	@Produces(MediaType.APPLICATION_JSON)
	public Response listarEmpregados() {
```

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

        @POST
        @Consumes(MediaType.APPLICATION_JSON)
        @Produces(MediaType.APPLICATION_JSON)
        public Response incluirEmpregado(Empregado emp) {
                UUID uuid = UUID.randomUUID();
                emp.setId(uuid.toString());
                empregados.put(uuid.toString(), emp);
                return Response.ok(emp).build();
        }

        @PUT
        @Path("{id}")
        @Consumes(MediaType.APPLICATION_JSON)
        @Produces(MediaType.APPLICATION_JSON)
        public Response alterarEmpregado(@PathParam("id") String
id, Empregado emp) {
                if (empregados.containsKey(id)) {
                        emp.setId(id);
                        empregados.put(id, emp);
                        return Response.ok().entity(emp).build();
                } else {
                        return Response.status(404).build();
                }
        }

        @DELETE
        @Path("{id}")
        @Consumes(MediaType.APPLICATION_JSON)
        @Produces(MediaType.APPLICATION_JSON)
        public Response excluirEmpregado(@PathParam("id") String
id) {
                if (empregados.containsKey(id))
{
                        empregados.remove(id);
                        return Response.ok().build();
```

```
                } else {
                        return Response.status(404).build();
                }
        }

}
```

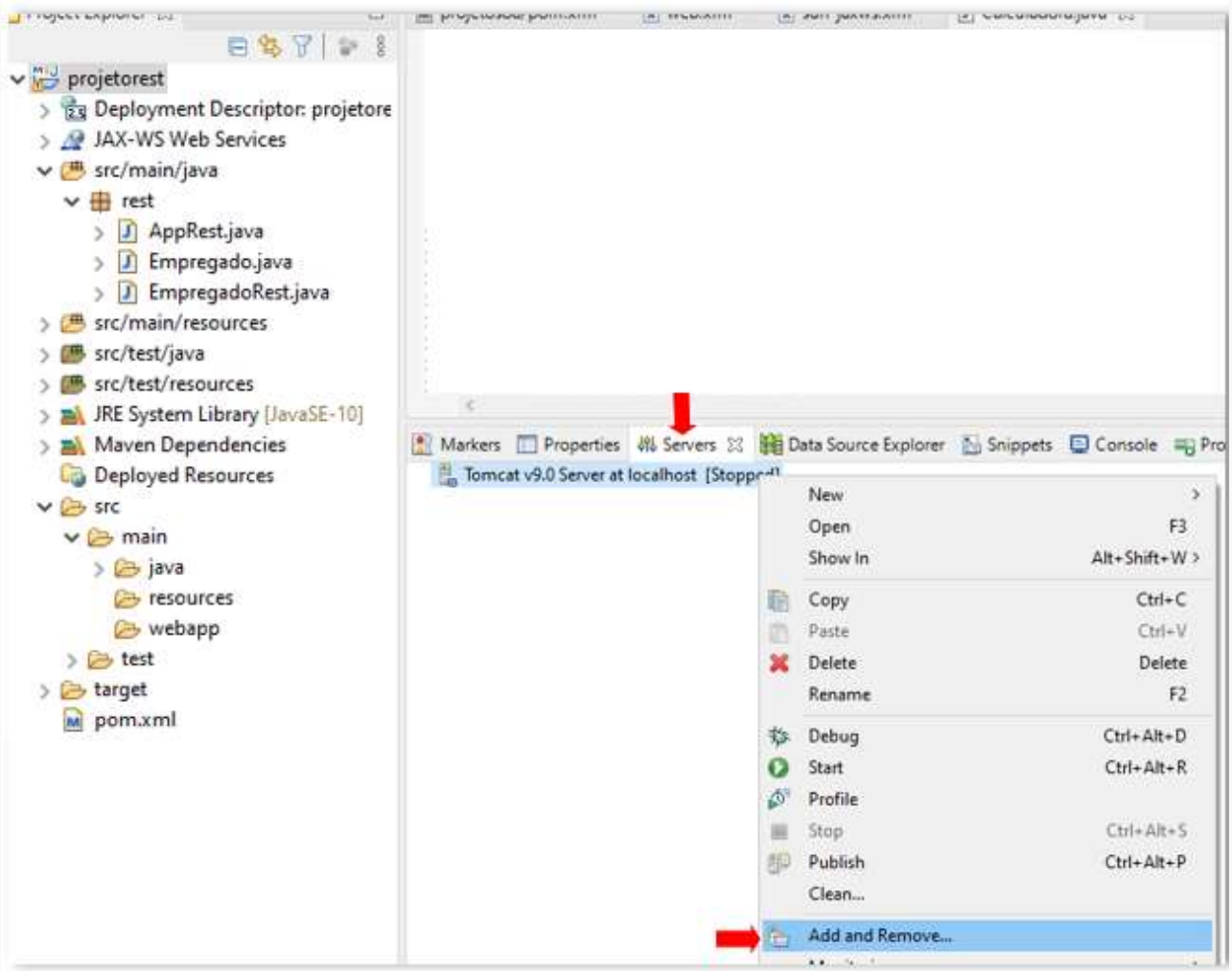
A classe **AppRest** terá informações importantes para descrição do serviço.

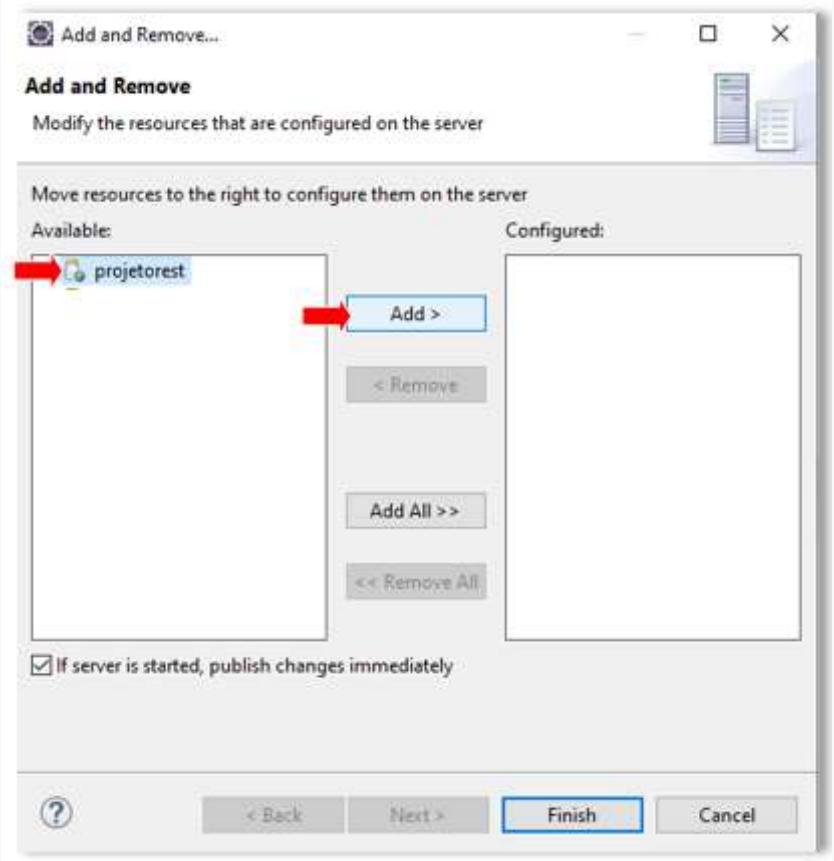
```
package rest;

import java.util.HashSet;
import java.util.Set;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

import io.swagger.jaxrs.config.BeanConfig;
import io.swagger.jaxrs.listing.ApiListingResource;
import io.swagger.jaxrs.listing.SwaggerSerializers;

@ApplicationPath("/rest")
public class AppRest extends Application {

        public AppRest() {
                BeanConfig conf = new BeanConfig();
                conf.setTitle("Projeto REST");
                conf.setDescription("Projeto REST");
                conf.setVersion("1.0.0");
                conf.setHost("localhost:8080");
                conf.setBasePath("/projetorest/rest");
                conf.setSchemes(new String[] { "http" });
                conf.setResourcePackage("rest");
                conf.setScan(true);
        }

        @Override
        public Set<Class<?>> getClasses() {
                Set<Class<?>> resources = new HashSet<>
();
```

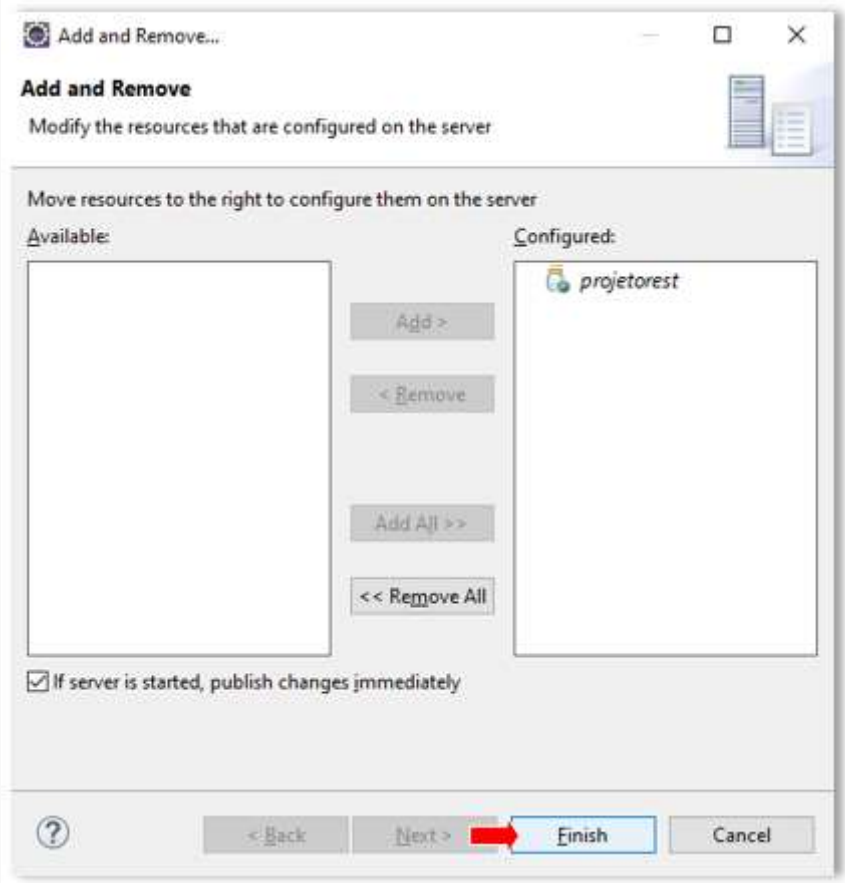
```
                resources.add(EmpregadoRest.class);

                // classes do swagger...
                resources.add(ApiListingResource.class);
                resources.add(SwaggerSerializers.class);
                return resources;
        }

}
```

O projeto está concluído!

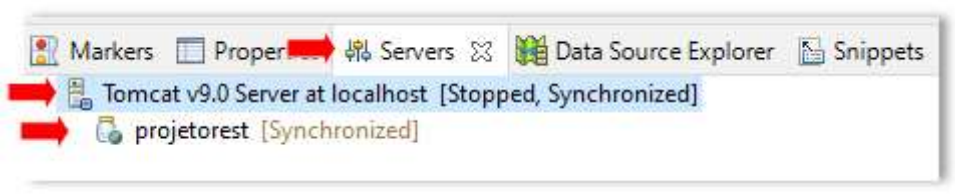
Por meio da aba **Servers**, adicione o projeto ao Apache Tomcat.



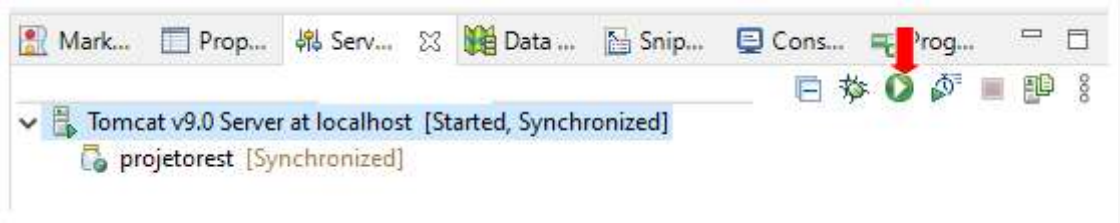




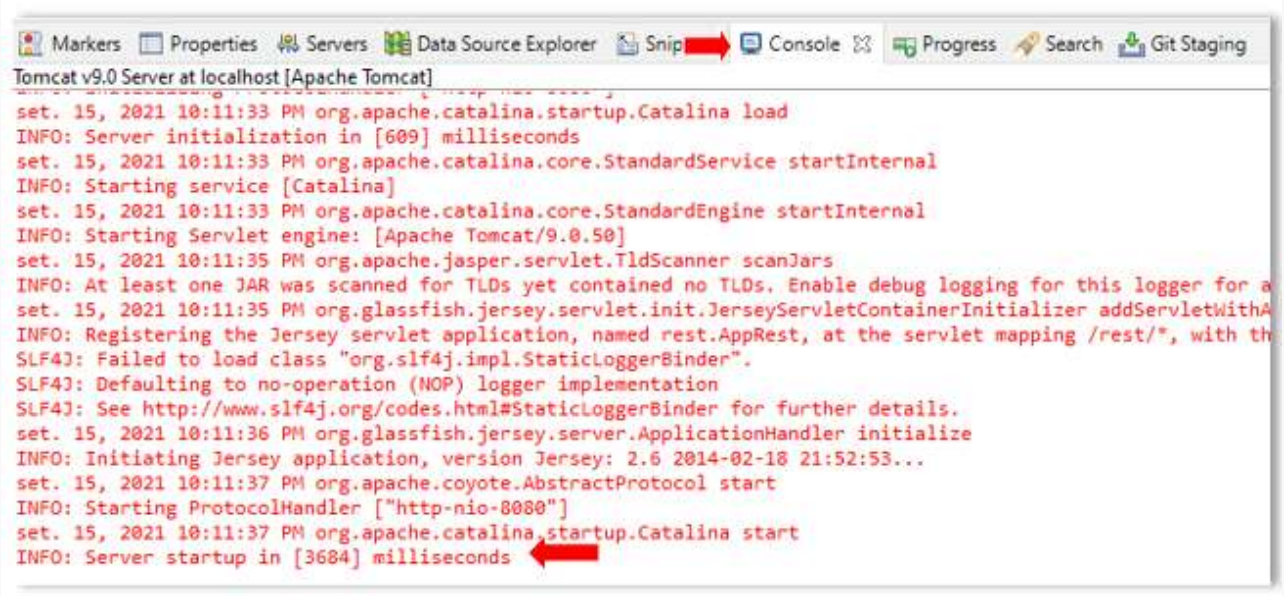
Observe que o projeto está associado ao Apache Tomcat.



Inicie o servidor.



Observe as mensagens de inicialização do Apache Tomcat na aba **Console**.



O serviço está disponível. Abra um navegador e informe a URL.

`http://localhost:8080/projetorest/rest/swagger.json`

Se você chegou até aqui, gostaríamos de dar parabéns, temos um *web service* REST funcional. Provavelmente, você deve ter algumas dúvidas, por isso temos a videoaula para revisar o código-fonte. Nela, são explicados os conceitos que estão em cada classe, o arquivo **swagger.json** e como testar o serviço por meio da ferramenta SoapUI.

Conclusão

- O HTTP é importante para executar ações em aplicativos na *web*.
- Os quatro principais métodos HTTP (POST, PUT, GET e DELETE) são utilizados, respectivamente, para incluir, alterar, consultar e excluir dados.
- As respostas aos pedidos HTTP são representadas por faixas de números.
- Os cabeçalhos (*headers*) servem para complementar as informações de uma requisição HTTP.
- O *payload* é a parte da requisição HTTP que possui os dados dos recursos.
- A especificação JAX-RS fornece suporte para criação de serviços *web* REST.

REFERÊNCIAS

ERL, T. **SOA**: princípios de design de serviços. São Paulo: Prentice Hall, 2009.

SMARTBEAR. **SOAP vs REST**: what's the difference? 2020. Disponível em: https://smartbear.com/blog/soap-vs-rest-whats-the-difference. Acesso em: 21 dez. 2021.