



[www.devmedia.com.br](http://www.devmedia.com.br)

[versão para impressão]

Link original: <http://www.devmedia.com.br/articles/viewcomp.asp?comp=33273>

# Criando um CRUD RESTful com Jersey, JPA e MySQL

Aprenda neste artigo a implementar serviços para um CRUD RESTful para cadastro de clientes.

---

## Fique por dentro

Este artigo é útil por demonstrar, passo a passo, como desenvolver um serviço RESTful com as operações de um CRUD para um cadastro de clientes totalmente web através de um projeto Java EE com Maven. Para que você possa compreender e aprender como utilizá-las, este projeto irá envolver uma gama de tecnologias da plataforma Java com o objetivo de criar uma solução de qualidade e escalável.

Para isso, será apresentada a construção de um DER para o banco de dados MySQL seguido pelo desenvolvimento de um serviço REST utilizando o Jersey, implementação de referência da especificação JAX-RS.

Além disso, será utilizada a especificação JPA e o Hibernate como ferramenta ORM para fazer o mapeamento objeto relacional entre as tabelas do banco de dados e as classes Java.

Ao final, você saberá como implementar seus primeiros serviços web, recurso cada vez mais comum no mercado de software, que busca soluções capazes de prover diferentes interfaces com o usuário e de fácil integração.

Durante muito tempo os web services baseados em SOAP foram praticamente a única solução para a comunicação e implementação de sistemas distribuídos.

Devido a isso e visando a qualidade dos serviços, essa opção passou por várias melhorias ao longo dos anos, principalmente relacionadas à segurança.

No entanto, os web services SOAP acabaram ficando complexos, de difícil implementação e com custos elevados de adoção. A partir de então, a opção por esse tipo de web service passou a ser inviável em alguns cenários, seja pelo custo, por recursos de hardware e de software ou mesmo pelo grande consumo de banda da rede, principalmente por parte dos dispositivos móveis, que ainda não têm uma conexão de alta velocidade a preços acessíveis.

O protocolo SOAP necessita de uma série de parâmetros e configurações no formato XML para viabilizar a troca de dados entre cliente e servidor e isso torna as mensagens longas tanto para tráfego na rede quanto para o dispositivo cliente processar a resposta.

O uso de web services surge da necessidade de se ter uma aplicação distribuída e escalável, necessidade esta que se espalha pelos mais diversos setores de negócio por todo o mundo, seja para integrar sistemas em diferentes plataformas, como os dispositivos móveis, seja para conectar sistemas web a sistemas legados, expor um canal de comunicação para clientes ou parceiros, dentre outras necessidades.

Neste cenário, com o intuito de facilitar a comunicação entre sistemas, podemos fazer uso do protocolo HTTP e usufruir do padrão arquitetural REST para implementar os serviços web, simplificando assim a troca de dados entre cliente e servidor.

Como um grande diferencial, o REST suporta os principais formatos para comunicação e troca de informações (JSON e XML), popularmente utilizados no desenvolvimento de sistemas distribuídos e em outras aplicações.

Conhecido por ser um estilo híbrido derivado de vários estilos arquiteturais baseados em rede, o REST tem como idealização e pilar a implementação de serviços web baseados no protocolo HTTP. O termo surgiu nos anos 2000, na dissertação do coautor do protocolo HTTP, Dr. Roy Thomas Fielding, para obtenção do título de PhD, com a dissertação "*Architectural Styles and the Design of Network-based Software Architectures*".

Neste artigo iremos desenvolver um serviço RESTful responsável por viabilizar um cadastro de clientes com todas as funcionalidades de um CRUD. O desenvolvimento desse cadastro consiste na implementação de um web service para que um cliente possa consumir e realizar as operações básicas de acesso ao banco de dados.

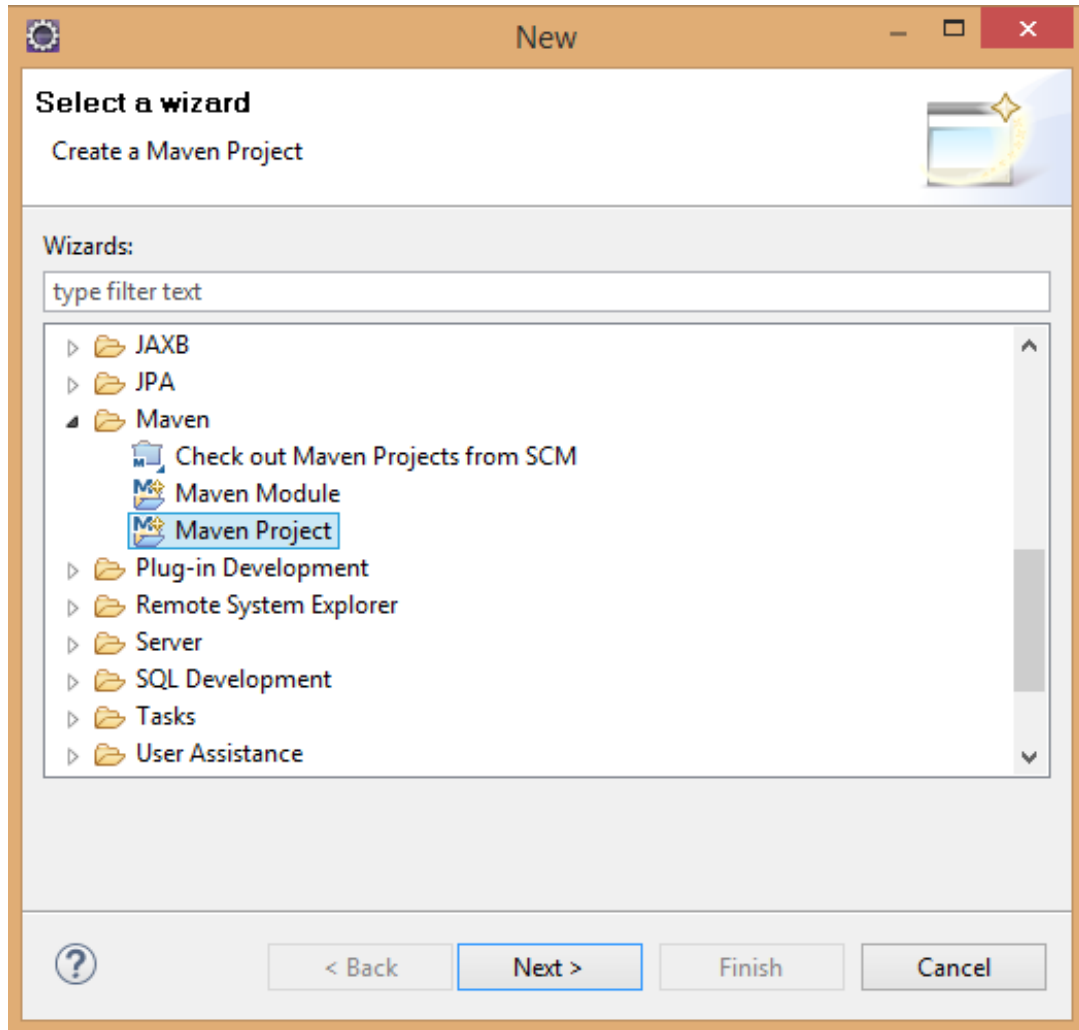
## Instalando o Eclipse Luna

Para o desenvolvimento do projeto "cadastro de clientes" iremos utilizar o Eclipse Luna. Sendo assim, baixe esta versão na página do Eclipse e então descompacte o arquivo em um diretório de sua preferência.

Ao executar esta IDE pela primeira vez é solicitado ao usuário que informe um diretório para servir como ambiente de trabalho. O workspace nada mais é do que uma pasta adotada pelo Eclipse para salvar os projetos que ele está gerenciando.

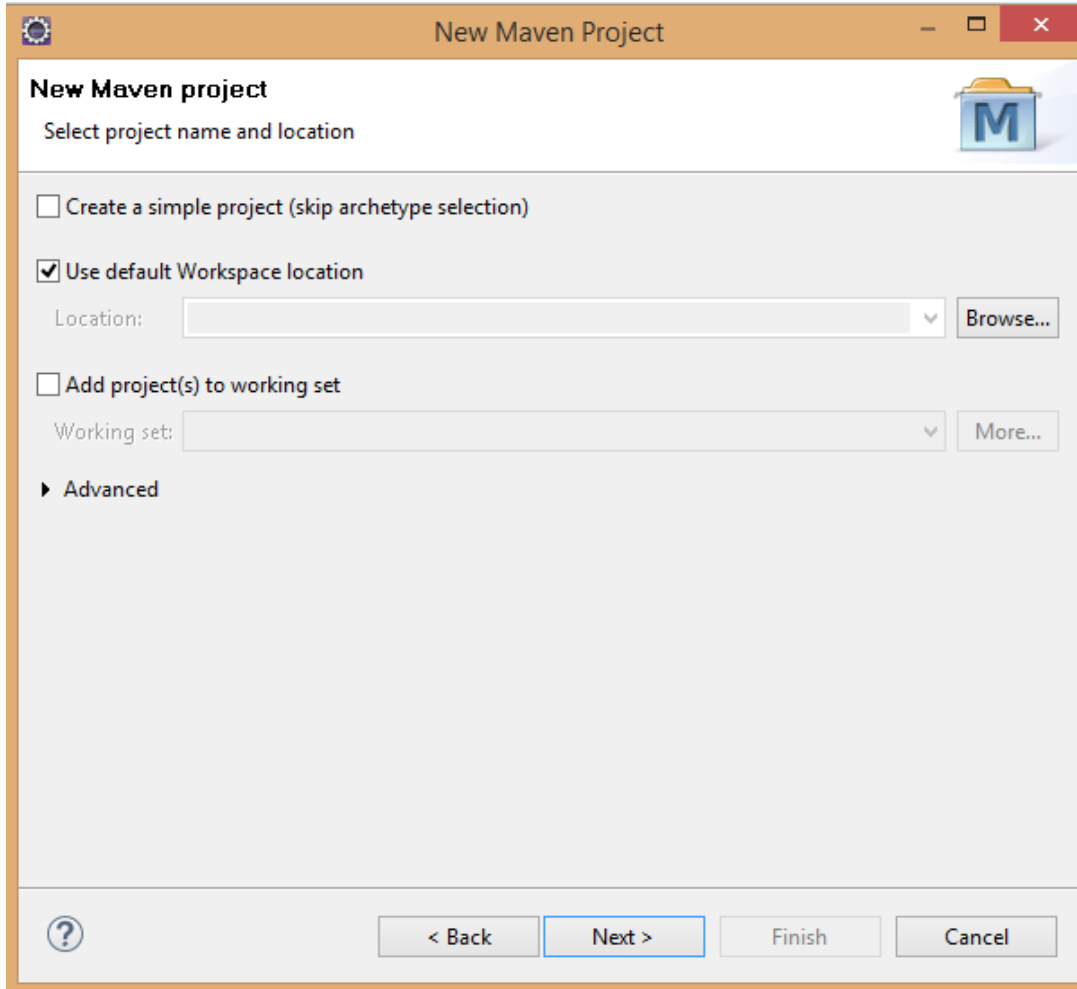
## Criando o projeto com Eclipse e Maven

Para criar o projeto no Eclipse, clique no menu *File > New > Other*. Logo após, será exibida uma nova janela, conforme a **Figura 1**, para que seja selecionado o wizard que auxiliará na criação do projeto. Neste caso, selecione a opção *Maven > Maven Project* e clique em *Next*.



**Figura 1.** Selecionando o wizard Maven Project.

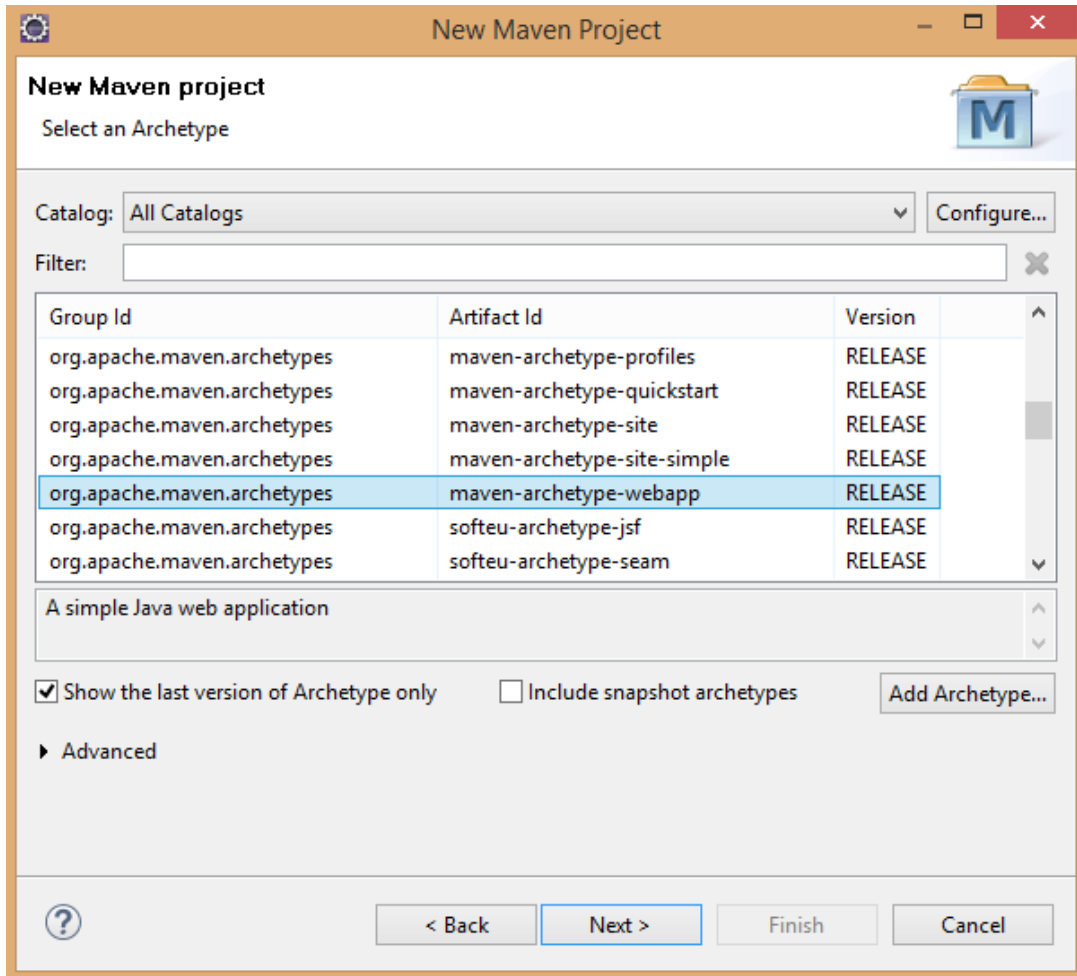
Na próxima tela, de criação do projeto Maven, deixe marcada a opção *Use default workspace location* para que seja utilizado o workspace configurado ao executar o Eclipse pela primeira vez. Em seguida, clique mais uma vez em *Next* (vide **Figura 2**).



**Figura 2.** Setando o workspace para o projeto.

Agora, conforme apresentado na **Figura 3**, deve ser selecionado o Archetype do Maven a ser utilizado para criar o projeto. Neste caso, selecione a opção *maven-archetype-webapp*, pois iremos criar um projeto Java EE.

O Archetype é uma espécie de template que viabiliza a criação de projetos com base em uma tecnologia ou especificação, como JSF, JPA, Spring, Struts, dentre outras. Feito isso, clique em *Next*.



**Figura 3.** Selecionando o Archetype do Maven para o projeto cadastro de clientes.

A última tela do wizard, apresentada na **Figura 4**, mostra alguns campos que devem ser preenchidos para finalizar a criação do projeto. Nela, deve ser informado o group id (geralmente informa-se o site da organização), o id do artefato (identificador do projeto), a versão deste e o nome completo do pacote base.

**New Maven project**  
Specify Archetype parameters

Group Id:

Artifact Id:

Version:

Package:

Properties available from archetype:

Name	Value

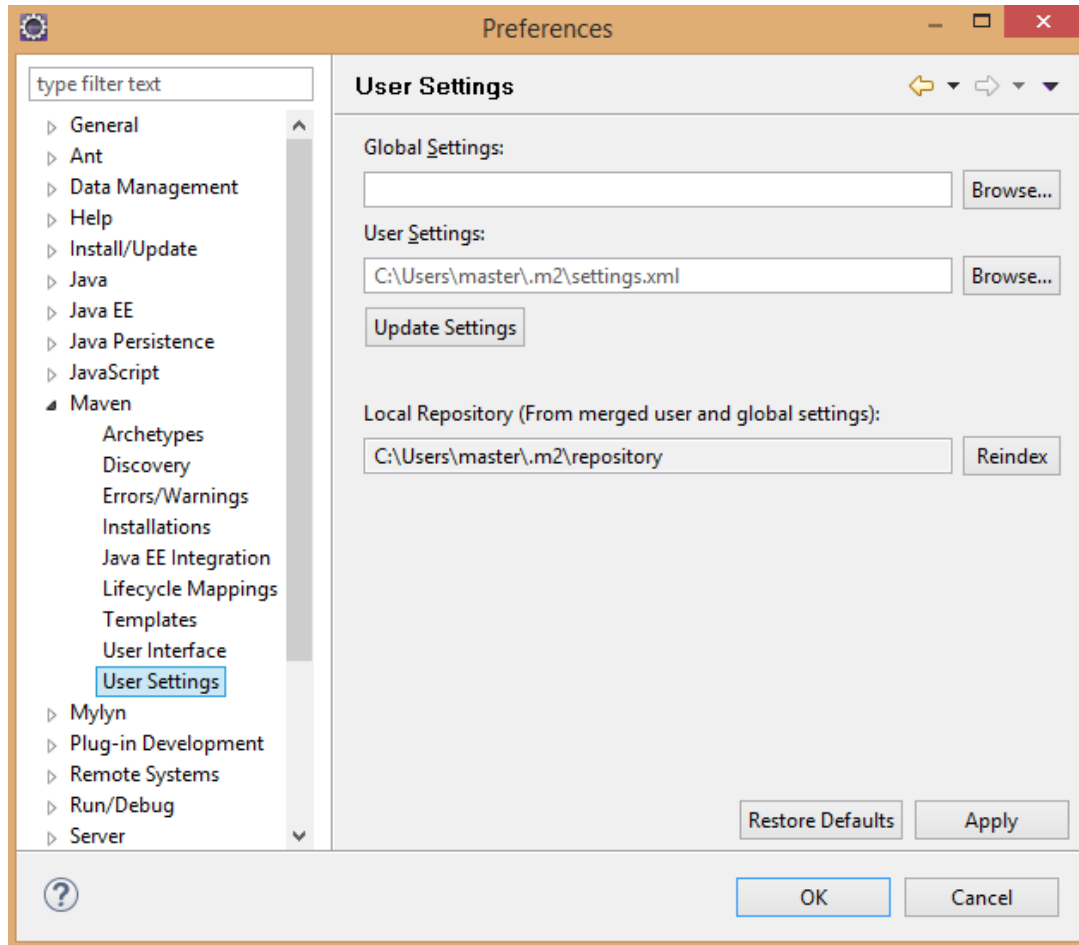
► Advanced

**Figura 4.** Definição dos dados para identificação do projeto.

Após clicar em *Finish*, em algumas ocasiões pode ocorrer um erro de acesso à pasta do repositório local do Maven, informando que não foi possível definir esse repositório. Para solucioná-lo você deve deletar a pasta *repository* para que o Eclipse possa criá-la novamente com as permissões de acesso corretas.

O local do repositório pode ser verificado na opção *Local Repository*, conforme apresentado na **Figura 5**, nas preferências do Eclipse.

O repositório nada mais é que uma pasta na máquina local onde o Maven irá baixar de seu servidor ou servidor personalizado todas as dependências do projeto, como frameworks, plugins e bibliotecas.



**Figura 5.** Definindo a nova pasta repository.

## Configurando as dependências e plugins no pom.xml

Para que possamos construir o projeto será necessário utilizar várias bibliotecas Java, referentes ao Jersey, Hibernate, MySQL e plugin do Tomcat. Sendo assim, devemos informar essas bibliotecas e suas respectivas versões no *pom.xml*.

Feito isso, o Maven se responsabilizará por baixá-las e gerenciá-las a partir de então. Na **Listagem 1** é apresentado o código do arquivo *pom.xml* do nosso projeto.

**Listagem 1.** Arquivo pom.xml com as dependências do projeto.

```

01 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
02   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
03   <modelVersion>4.0.0</modelVersion>
04   <groupId>br.com.devmedia</groupId>
05   <artifactId>crud_cadastro_cliente</artifactId>
06   <packaging>war</packaging>
07   <version>0.0.1-SNAPSHOT</version>
08   <name>crud_cadastro_cliente Maven Webapp</name>
09   <url>http://maven.apache.org</url>
10   <dependencies>
11   <dependency>
12     <groupId>junit</groupId>
13     <artifactId>junit</artifactId>

```

```

14     <version>3.8.1</version>
15     <scope>test</scope>
16 </dependency>
17 <dependency>
18     <groupId>com.sun.jersey</groupId>
19     <artifactId>jersey-server</artifactId>
20     <version>1.8</version>
21 </dependency>
22 <dependency>
23     <groupId>com.sun.jersey</groupId>
24     <artifactId>jersey-json</artifactId>
25     <version>1.8</version>
26 </dependency>
27 <dependency>
28     <groupId>org.hibernate</groupId>
29     <artifactId>hibernate-validator</artifactId>
30     <version>4.2.0.Final</version>
31 </dependency>
32 <dependency>
33     <groupId>org.hibernate.common</groupId>
34     <artifactId>hibernate-commons-annotations</artifactId>
35     <version>4.0.1.Final</version>
36     <classifier>tests</classifier>
37 </dependency>
38 <dependency>
39     <groupId>org.hibernate.javax.persistence</groupId>
40     <artifactId>hibernate-jpa-2.0-api</artifactId>
41     <version>1.0.1.Final</version>
42 </dependency>
43 <dependency>
44     <groupId>org.hibernate</groupId>
45     <artifactId>hibernate-entitymanager</artifactId>
46     <version>4.0.1.Final</version>
47 </dependency>
48 <dependency>
49     <groupId>mysql</groupId>
50     <artifactId>mysql-connector-java</artifactId>
51     <version>5.1.6</version>
52     <scope>compile</scope>
53 </dependency>
54 </dependencies>
55 <build>
56     <finalName>crud_cadastro_cliente</finalName>
57     <plugins>
58         <plugin>
59             <groupId>org.apache.tomcat.maven</groupId>
60             <artifactId>tomcat7-maven-plugin</artifactId>
61             <version>2.0</version>
62             <configuration>
63                 <path></path>
64                 <port>8080</port>
65             </configuration>
66         </plugin>
67     </plugins>
68 </build>
69 </project>

```

Vejamos a seguir algumas considerações sobre a listagem:



- **Linhas 4 a 8:** Neste trecho são definidas algumas informações referentes ao projeto para que o Maven possa realizar o controle do ciclo de vida do desenvolvimento. Assim sendo, é informado o identificador do grupo, o identificador do artefato, a versão e o nome do projeto;
- **Linhas 11 a 16:** Aqui foi inserida uma dependência ao JUnit para testes de unidade no projeto. O Maven define a mesma automaticamente no ato da criação do projeto;
- **Linhas 17 a 21:** Neste bloco consta a dependência ao servidor Jersey, framework responsável por tratar as requisições realizadas ao serviço. Seu código assume o papel de Servlet Container para receber e tratar todas as requisições realizadas via protocolo HTTP. Neste projeto iremos implementar as requisições HTTP com os verbos GET, POST, PUT e DELETE para as respectivas operações de um CRUD. Assim será possível inserir e realizar a manutenção dos dados no banco MySQL;
- **Linhas 22 a 26:** Neste bloco é definida a dependência para o suporte à troca de dados no formato JSON. Desta forma será possível enviar e receber dados no formato JSON entre as requisições e respostas do cliente ao servidor;
- **Linhas 27 a 47:** Neste intervalo são informadas todas as dependências do Hibernate. O Hibernate é um framework que viabiliza o mapeamento objeto relacional entre as tabelas do banco de dados e as classes Java. Lembre-se que as classes Java representam as entidades do mundo real no sistema. É importante frisar ainda que embora tenhamos adotado o Hibernate, programaticamente iremos utilizar as interfaces da especificação JPA para acesso ao banco;
- **Linhas 48 a 53:** Aqui é definida a dependência da biblioteca do MySQL. O Hibernate fará uso desta para acessar o banco de dados.

Definidas as dependências, vamos configurar um servidor web que rode aplicações Java EE. Para isso, podemos usar o WildFly, GlassFish, dentre outros, porém, neste artigo será adotado o Tomcat 7.

Como a Apache disponibiliza o mesmo através de um plug-in para projetos desenvolvidos com o Maven, a sua instalação no Eclipse é bastante simples. Basta adicionar o bloco de código referente ao plug-in do Tomcat no *pom.xml* e executar o projeto definindo o campo *Goals* com o valor *tomcat7:run*, como podemos verificar na janela *Run Configurations* do Eclipse (vide **Figura 6**).

Feito isso o Maven irá baixar todos os arquivos necessários para o repositório local e depois irá levantar o servidor no endereço *localhost:8080*. As linhas 68 a 78 mostram como configurar esse plug-in, onde informamos o Group Id, Artifact Id, version e o path para formar a URL a ser acessada pelas requisições HTTP.

Realizadas as configurações, execute o projeto para verificar o resultado no browser. Durante a compilação do projeto você pode acompanhar o download do plug-in do Tomcat 7 através do console do Eclipse. Ao final será apresentada a URL do servidor. A partir de então podemos informar a mesma no browser para verificar a aplicação rodando, como demonstra a **Figura 7**.

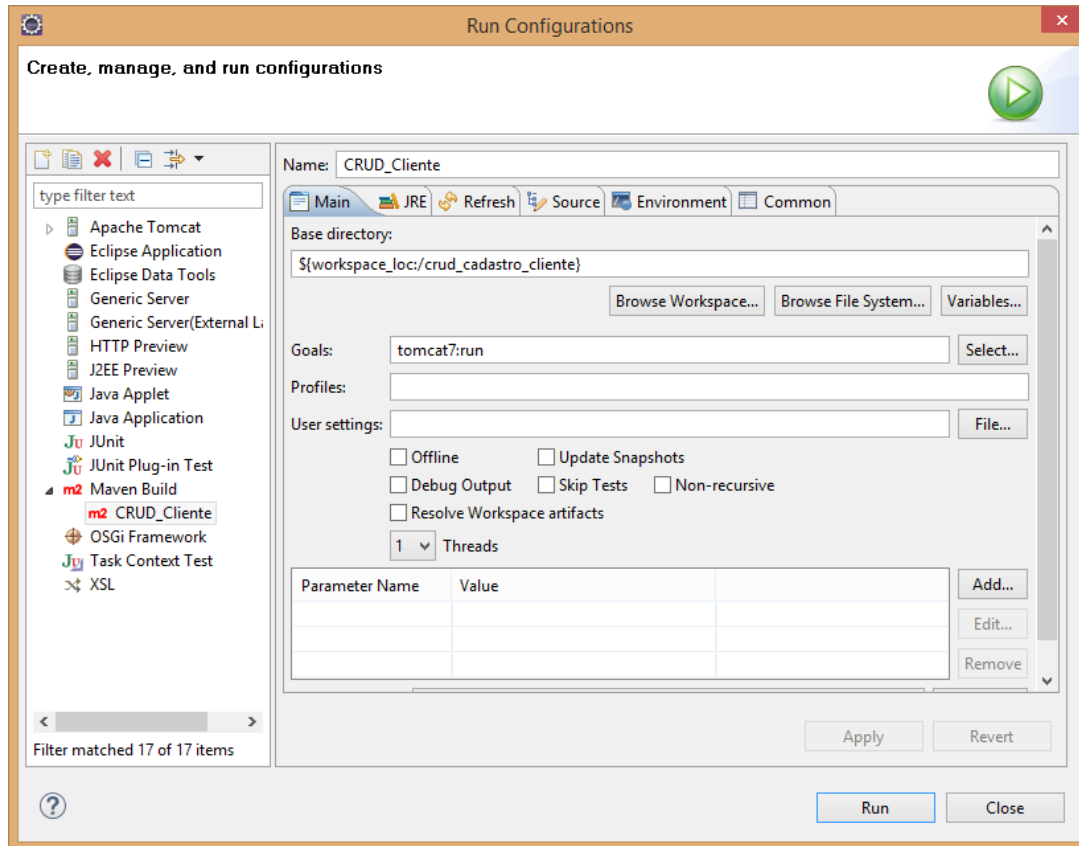
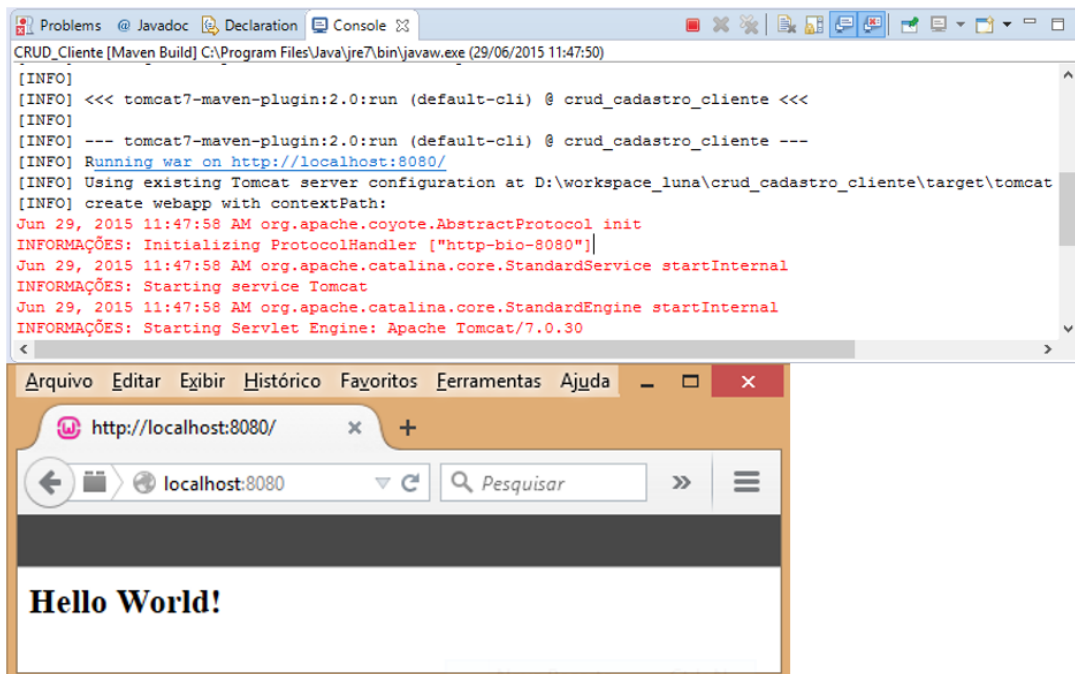


Figura 6. Configurando o build do projeto.



[abrir imagem em nova janela](#)

Figura 7. Projeto cadastro de clientes em execução no browser.

## Diagrama Entidade Relacionamento da aplicação

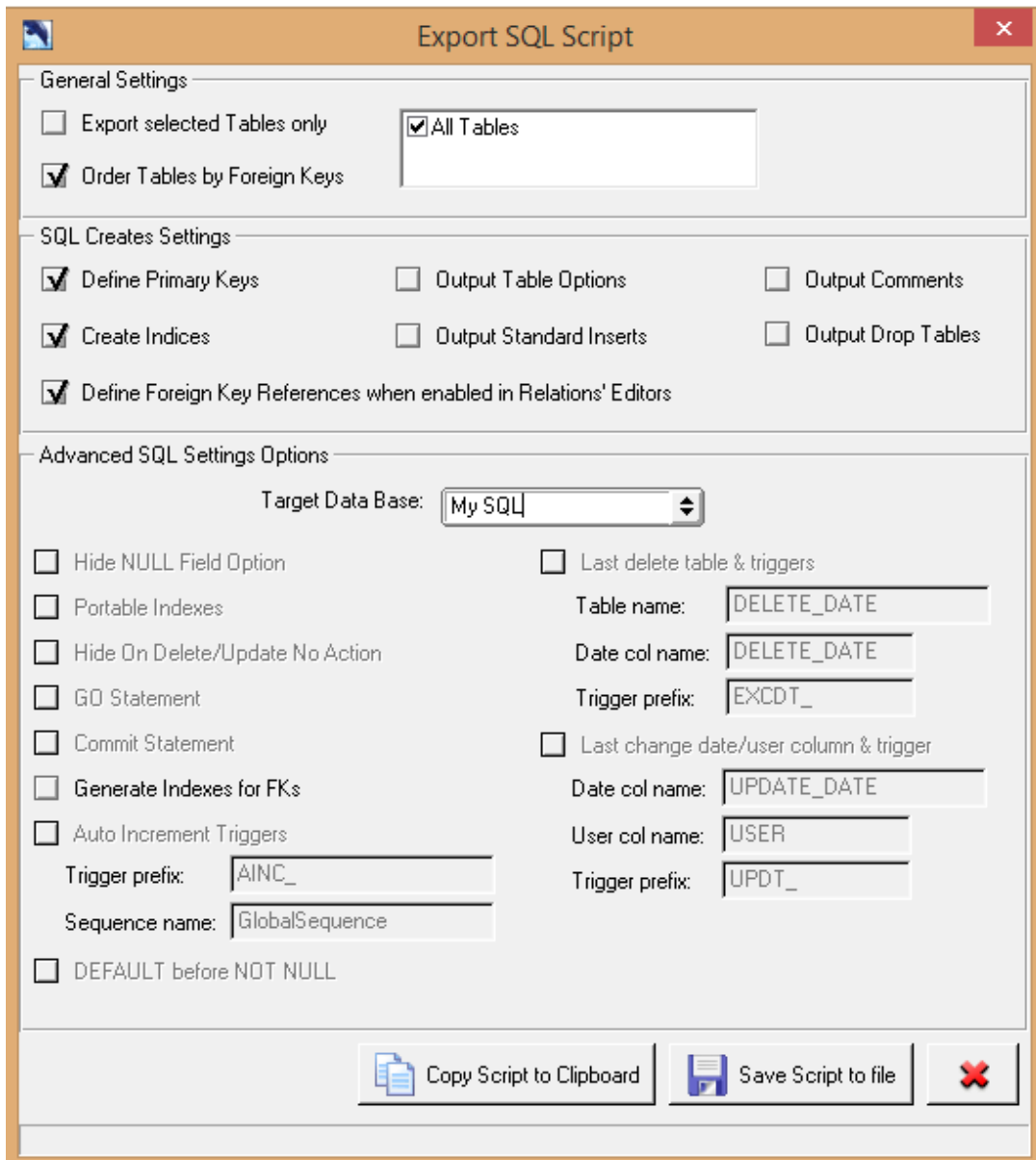
Para desenvolver o Diagrama Entidade Relacionamento (DER) da aplicação e posteriormente gerar a(s) tabela(s) no banco de dados, iremos utilizar a ferramenta CASE DBDesigner Fork, uma solução gratuita que gera scripts SQL para diversos SGBDs, como MySQL, SQL Server, Oracle, SQLite, PostgreSQL, dentre outros. Você pode realizar o download do DBDesigner através do endereço indicado na seção **Links**.

A **Figura 8** apresenta o DER da nossa aplicação sendo composto por apenas uma tabela, de nome *cliente*, responsável por armazenar os dados dos clientes a serem cadastrados.

Para gerar o script SQL para o MySQL, clique no menu *File > Export > SQL Create Script*. Feito isso, será aberta uma nova tela conforme apresentado na **Figura 9**. Então, selecione a opção *MySQL* no *Target Data Base* e clique no botão *Copy script to Clipboard* para copiar o script. Logo após, abra alguma ferramenta de gerência do banco de dados, como o MySQL Workbench, crie um banco de dados chamado "bd\_cliente" e execute o script da **Listagem 2**, gerado pelo DBDesigner Fork.

cliente	
id_cliente	INTEGER
nome	VARCHAR(100)
cpf_cnpj	VARCHAR(14)
rg	VARCHAR(20)
endereco	VARCHAR(100)
bairro	VARCHAR(60)
cidade	VARCHAR(60)
estado	VARCHAR(50)
telefone	VARCHAR(12)
email	VARCHAR(50)
data_cadastro	DATE

**Figura 8.** DER do banco de dados cadastro de clientes.



**Export SQL Script**

**General Settings**

Export selected Tables only     All Tables

Order Tables by Foreign Keys

**SQL Creates Settings**

Define Primary Keys     Output Table Options     Output Comments

Create Indices     Output Standard Inserts     Output Drop Tables

Define Foreign Key References when enabled in Relations' Editors

**Advanced SQL Settings Options**

Target Data Base:

Hide NULL Field Option     Last delete table & triggers

Portable Indexes    Table name:

Hide On Delete/Update No Action    Date col name:

GO Statement    Trigger prefix:

Commit Statement     Last change date/user column & trigger




Generate Indexes for FKs    Date col name:

Auto Increment Triggers    User col name:

Trigger prefix:     Trigger prefix:

Sequence name:

DEFAULT before NOT NULL

 Copy Script to Clipboard     Save Script to file    

**Figura 9.** Gerando script SQL.

**Listagem 2.** Script SQL para criar a tabela cliente.

```
CREATE TABLE cliente (
  id_cliente INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  nome VARCHAR(100) NULL ,
  cpf_cnpj VARCHAR(14) NULL ,
  rg VARCHAR(20) NULL ,
  endereco VARCHAR(100) NULL ,
  bairro VARCHAR(60) NULL ,
  cidade VARCHAR(60) NULL ,
  estado VARCHAR(50) NULL ,
  telefone VARCHAR(12) NULL ,
  email VARCHAR(50) NULL ,
  data_cadastro DATE NULL ,
  PRIMARY KEY(id_cliente));
```

## Criando a entidade cliente

Para tornar possível a manipulação de clientes na aplicação é necessária ter uma classe que viabilize essa abstração. A partir dela podemos receber os dados do cliente armazenados no banco de dados, exibir na interface do usuário e vice-versa.

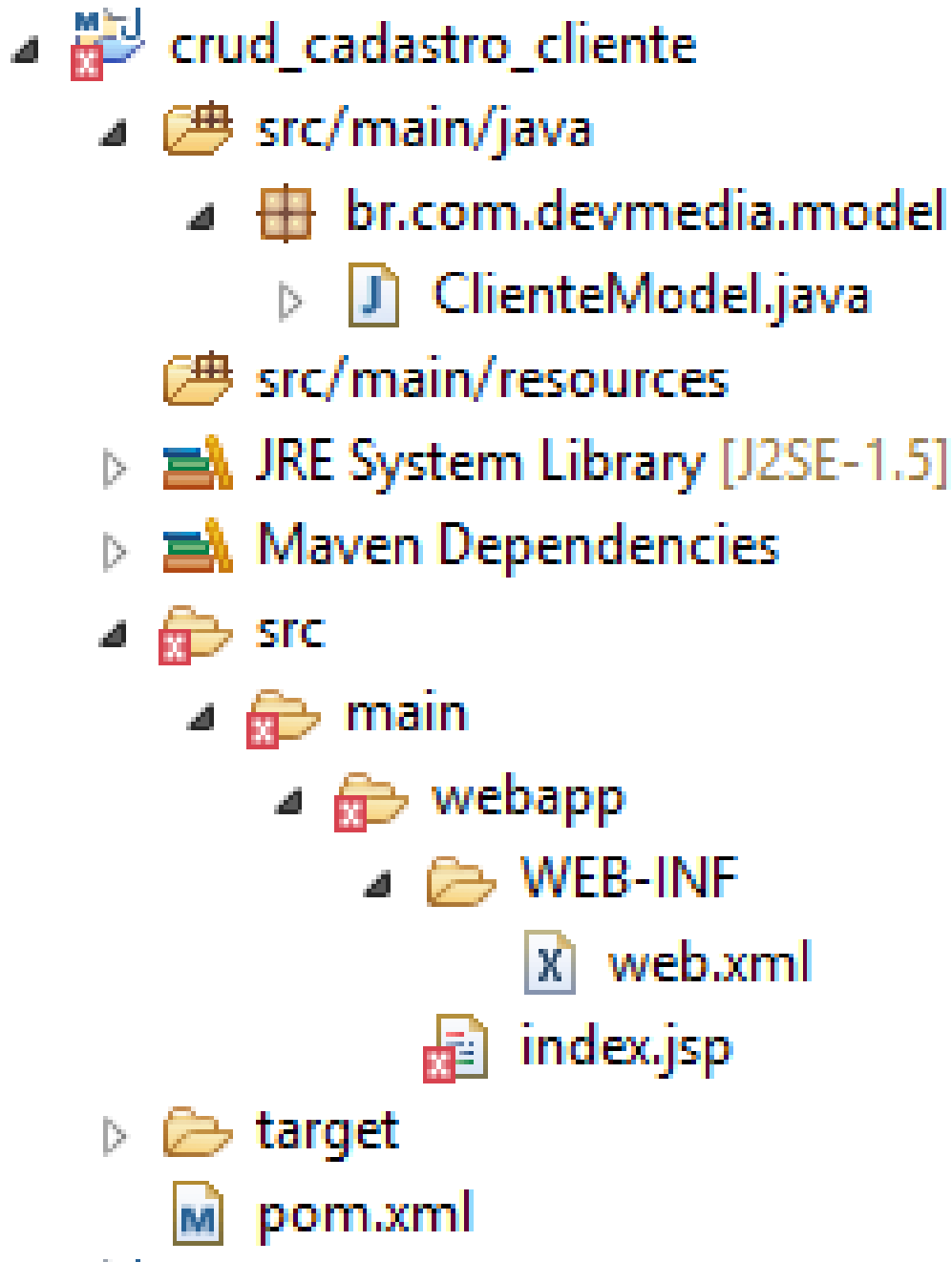
Sendo assim, vamos criar uma classe de nome **ClienteModel** para representar a entidade cliente. Nesta classe declararemos os atributos referentes às colunas da tabela *cliente* do banco de dados, processo este que é conhecido como mapeamento objeto relacional. Para isso, primeiro criaremos uma pasta chamada *java* dentro da pasta *main* do projeto.

É importante seguir estas instruções porque caso você crie uma pasta com um nome diferente de *java*, o Jersey e o Hibernate podem não encontrar as classes **ClienteModel** e **ClienteService**, que serão criadas para expor o serviço REST com as operações do CRUD.

Para criar a pasta *java*, clique com o botão direito na pasta *main* e depois na opção *New > Folder*. Em seguida, defina o nome como *java* e clique em *Finish*.

Agora, clique com o botão direito na pasta *src/main/java* e depois, ao selecionar a opção *New > Class*, defina o nome da classe como **ClienteModel**, no pacote **br.com.devmedia.model**, e clique em *Finish*.

Realizado este procedimento, teremos o projeto com a estrutura apresentada na **Figura 10**.



**Figura 10.** Estrutura do projeto.

Criada a classe **ClienteModel**, você deve implementá-la conforme o código apresentado na **Listagem 3**.

**Listagem 3.** Código da classe `ClienteModel`.

```

01 package br.com.devmedia.model;
02
03 import java.util.Date;
04 import javax.persistence.Column;
05 import javax.persistence.Entity;
06 import javax.persistence.GeneratedValue;
07 import javax.persistence.GenerationType;
08 import javax.persistence.Id;
09 import javax.persistence.Table;

```

```
10 import javax.persistence.Temporal;
11 import javax.persistence.TemporalType;
12
13 @Entity
14 @Table(name="cliente")
15 public class ClienteModel {
16     @Id
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     int id_cliente;
19
20     @Column(name="nome")
21     String nome;
22
23     @Column(name="cpf_cnpj")
24     String cpf_cnpj;
25
26     @Column(name="rg")
27     String rg;
28
29     @Column(name="endereco")
30     String endereco;
31
32     @Column(name="bairro")
33     String bairro;
34
35     @Column(name="cidade")
36     String cidade;
37
38     @Column(name="estado")
39     String estado;
40
41     @Column(name="email")
42     String email;
43
44     @Column(name="data_cadastro")
45     @Temporal(TemporalType.TIMESTAMP)
46     Date data_cadastro;
47
48     public int getId_cliente() {
49         return id_cliente; }
50
51     public void setId_cliente(int id_cliente) {
52         this.id_cliente = id_cliente; }
53
54     public String getNome() {
55         return nome; }
56
57     public void setNome(String nome) {
58         this.nome = nome; }
59
60 }
```

Como podemos verificar, esse código tem algumas particularidades e anotações da JPA que merecem destaque. Assim, a seguir analisamos os detalhes dessa implementação:

- **Linha 13:** Nesta linha é informada a anotação **@Entity** da JPA. Ela é responsável por definir que a classe **ClienteModel** é uma estrutura de mapeamento objeto relacional e corresponde a uma referência a uma tabela na base de dados;

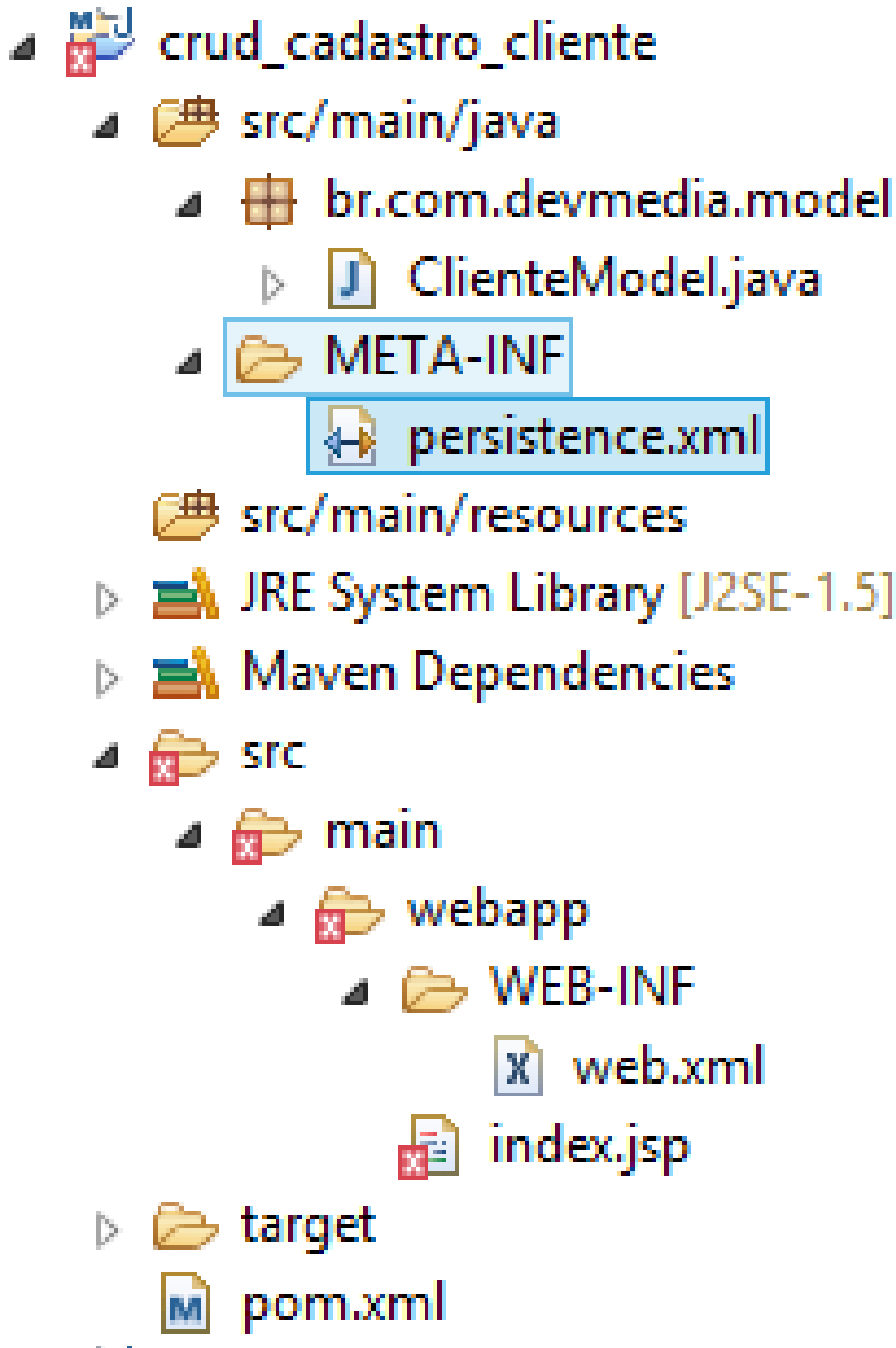


- **Linha 14:** Aqui é definida a anotação **@Table** com a propriedade **name** especificada com o valor "cliente". Deste modo, informamos que a classe **ClienteModel** corresponde à tabela de nome *cliente* do banco de dados;
- **Linhas 16 a 18:** Neste intervalo foi definida a propriedade **id\_cliente** com as anotações **@Id**, para especificar que é chave primária, e **@GeneratedValue**, para especificar que o valor da chave primária deve ser gerado automaticamente pelo banco de dados;
- **Linhas 20 a 42:** Neste intervalo são determinadas as demais propriedades da classe **ClienteModel**. Perceba que todas são anotadas com **@Column** para informar que representam colunas na tabela *cliente* da base de dados;
- **Linha 45:** Esta linha expõe a anotação **@Temporal** para a propriedade **data\_cadastro**. Assim, é definido que este campo irá trabalhar com valores no formato data;
- **Linhas 48 a 58:** Neste intervalo são implementados os métodos de acesso (getters e setters) de algumas propriedades da classe **ClienteModel**. Você pode gerar estes métodos automaticamente no Eclipse. Para isso, basta clicar com o botão direito na opção *Source* > *Generate Getters and Setters* e marcar todos os atributos para os quais deseja gerar os respectivos métodos.

## Criando e configurando o persistence.xml

Para que nossa aplicação de cadastro de clientes consiga realizar o acesso a dados, devemos criar o arquivo *persistence.xml*. Este possui as configurações utilizadas pela JPA para viabilizar a comunicação com o banco de dados. Tal arquivo deve ser criado na pasta de nome *META-INF*, que por sua vez deve ser criada dentro da pasta *java* para que as classes **EntityManagerFactory** e **EntityManager** da JPA possam encontrá-lo e assim fazer uso das configurações informadas.

Dito isso, crie a pasta *META-INF* dentro de *java* e depois crie o arquivo *persistence.xml*. Após este procedimento a estrutura de pastas do projeto deve estar semelhante à apresentada na **Figura 11**.



**Figura 11.** Estrutura de pastas para inclusão do arquivo persistence.xml.

Com o *persistence.xml* em mãos, podemos configurá-lo para informar a classe **ClienteModel**, o provedor de acesso a dados, o dialeto do SGBD e os dados para conexão com o banco de dados, ou seja, o endereço do banco, o usuário e senha. A **Listagem 4** mostra como deve ficar esse arquivo.

**Listagem 4.** Código do arquivo persistence.xml.

```

01 <persistence xmlns="http://java.sun.com/xml/ns/persistence"
02 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
04 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
05 version="2.0">
06 <persistence-unit name="app_crud_cliente">
07 <provider>org.hibernate.ejb.HibernatePersistence</provider>
08 <class>br.com.devmedia.model.ClienteModel</class>
09 <properties>
10 <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
11 <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost/bd_cliente" />
12 <property name="javax.persistence.jdbc.user" value="root" />
13 <property name="javax.persistence.jdbc.password" value="root" />
14 <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
15 </properties>
16 </persistence-unit>
17 </persistence>

```

Vejamos os detalhes da configuração do *persistence.xml*:

- **Linha 6:** Nesta linha é informado o **persistence-unit**, a base para iniciar a configuração do *persistence.xml*. Perceba que a propriedade **name**, na mesma linha, recebeu o valor **"app\_crud\_cliente"**. Portanto, é este nome que informaremos na configuração do **EntityManagerFactory** que iremos criar mais adiante no artigo, para que possa ser identificada a configuração do *persistence.xml* e assim criar o **EntityManager** para acesso ao banco de dados;
- **Linha 7:** Aqui é informado o provedor ORM que implementa o JPA, neste caso o Hibernate;
- **Linha 8:** Nesta linha é informada a classe **ClienteModel**, responsável pelo mapeamento com o banco de dados;
- **Linhas 9 a 15:** Neste bloco de código são informadas as propriedades de conexão com o banco de dados, como o driver do MySQL, a URL de acesso ao servidor do banco de dados seguida pelo nome do banco, o usuário e a senha.

## Criando o EntityManager

O próximo passo é criar a classe que tenha um método responsável por retornar um objeto do tipo **EntityManager**. É o **EntityManager** que disponibiliza todos os métodos que precisamos para conseguir acesso ao banco de dados e que veremos mais à frente.

Sendo assim, criaremos uma classe, chamada **JpaEntityManager**, para gerenciar a conexão com o banco de dados MySQL. Dentro da mesma devemos declarar dois objetos para setar a fábrica de objetos, isto é, setar **EntityManagerFactory** com as configurações do arquivo *persistence.xml*. Assim, podemos obter uma instância do **EntityManager** para, de fato, chamar seus métodos e realizar as operações do CRUD.

Portanto, crie a classe **JpaEntityManager** no pacote **br.com.devmedia.EntityManager**. Seu código fonte é apresentado na **Listagem 5**.

**Listagem 5.** Código da classe JpaEntityManager.

```

01 package br.com.devmedia.EntityManager;
02
03 import javax.persistence.EntityManager;
04 import javax.persistence.EntityManagerFactory;

```

```

05 import javax.persistence.Persistence;
06
07 public class JpaEntityManager {
08
09     private EntityManagerFactory factory = Persistence.createEntityManagerFactory("app_crud_cliente");
10     private EntityManager em = factory.createEntityManager();
11
12     public EntityManager getEntityManager(){
13         return em;
14     }
15 }

```

Como podemos verificar, ele é bem simples. Vejamos os seus detalhes:

- **Linha 9:** Nesta linha é criado um objeto chamado **factory** do tipo **EntityManagerFactory**. Este foi definido para receber uma instância do próprio **EntityManagerFactory**. Para isso, chamamos o método **createEntityManagerFactory()** da classe **Persistence**, que recebe como parâmetro o nome **"app\_crud\_cliente"**, nome este que foi informado no *persistence.xml*, na propriedade **persistence-unit**;
- **Linha 10:** Neste trecho foi criado mais um objeto, chamado **em** e do tipo **EntityManager**. O mesmo recebe uma instância do **EntityManager** através da chamada ao método **createEntityManager()** do objeto **factory** criado anteriormente;
- **Linhas 12 a 14:** Neste bloco foi definido um método chamado **getEntityManager()**. Este retorna um objeto **EntityManager** para realizar as operações do CRUD no banco de dados através de seus métodos.

## Definindo o servlet do Jersey no web.xml

Em toda aplicação Java EE precisamos criar um arquivo chamado *web.xml*, local onde podemos configurar o servlet que irá receber as requisições HTTP. Neste arquivo também são definidas as informações sobre o serviço REST, pacotes, classes do serviço, dentre outras configurações que fogem do escopo deste artigo.

O nosso *web.xml* deve ser configurado conforme o código da **Listagem 6**, onde são informados os dados da API do Jersey, isto é, onde é definido que o servlet do Jersey que será o responsável por receber as requisições HTTP referentes às chamadas ao serviço.

**Listagem 6.** Definição do servlet do Jersey no arquivo web.xml.

```

01 <!DOCTYPE web-app PUBLIC
02 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
03 "http://java.sun.com/dtd/web-app_2_3.dtd" >
04
05 <web-app>
06     <display-name>Archetype Created Web Application</display-name>
07
08     <servlet>
09         <servlet-name>CRUD Cadastro de Clientes</servlet-name>
10         <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
11         <init-param>
12             <param-name>com.sun.jersey.config.property.packages</param-name>
13             <param-value>br.com.devmedia.service</param-value>
14         </init-param>
15
16     <init-param>

```

```
17 <param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>
18 <param-value>true</param-value>
19 </init-param>
20
21 <load-on-startup>1</load-on-startup>
22 </servlet>
23
24 <servlet-mapping>
25 <servlet-name>CRUD Cadastro de Clientes</servlet-name>
26 <url-pattern>/apiREST/*</url-pattern>
27 </servlet-mapping>
28 </web-app>
```

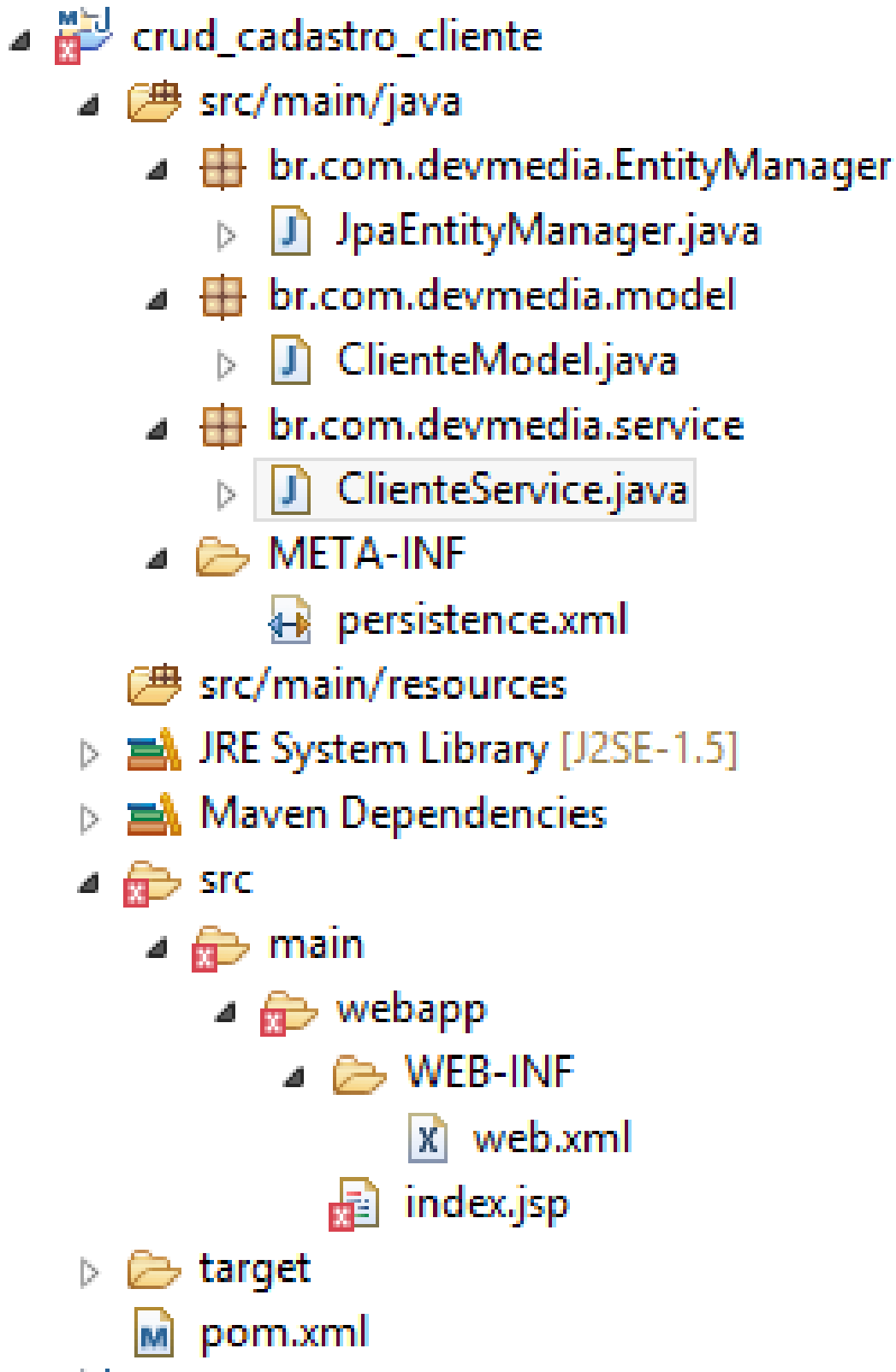
Para entender melhor esse código XML, vejamos sua análise:

- **Linha 10:** Nesta linha é definida a classe que representa o servlet da aplicação. Neste caso foi informado **ServletContainer**, do Jersey;
- **Linhas 11 a 14:** Neste bloco é informado o pacote que irá conter as classes que representam o serviço e contêm os métodos a serem expostos. Neste caso foi informado o pacote **br.com.devmedia.service**, que ainda vamos criar;
- **Linhas 16 a 19:** Neste bloco são definidos outros parâmetros para o serviço. Dessa vez para a API de JSON do Jersey, para que ela possa realizar o mapeamento que viabiliza a conversão das requisições no formato JSON em classes Java. O mesmo vale para as respostas do servidor ao cliente, na transformação objeto Java -> JSON;
- **Linhas 24 a 27:** Neste bloco é configurada a URL para acesso aos recursos do servidor, ou melhor, às operações do CRUD a serem expostas pelo serviço. Através da tag **url-pattern** é definido o caminho padrão de acesso às URLs do serviço, como: *http://localhost:8080/apiREST/cliente/cadastrar*.

## Implementando o serviço REST

Chegou o momento de implementar a classe que irá expor o serviço REST e conseqüentemente as operações CRUD para o cadastro de clientes. Para desenvolver esse serviço, crie uma classe denominada **ClienteService** no pacote **br.com.devmedia.service**. Após esse passo, a estrutura do projeto deve estar conforme a **Figura 12**. O código da classe **ClienteService** é apresentado na **Listagem 7**.

Observando essa imagem podemos notar uma organização na estrutura dos pacotes do projeto. Sendo assim, há um pacote para representar as classes que representam as entidades, outro para as classes de serviço e, por fim, o pacote que encapsula o acesso dados. Desta forma o projeto se torna bem estruturado e com responsabilidades bem definidas, facilitando possíveis atualizações e manutenções.



**Figura 12.** Estrutura do projeto após criar a classe ClienteService.

**Listagem 7.** Código do serviço REST – classe ClienteService.

```
01 package br.com.devmedia.service;
```

```
02
03 @Path("/cliente")
04 public class ClienteService {
05     private JpaEntityManager JPAEM = new JpaEntityManager();
06     private EntityManager objEM = JPAEM.getEntityManager();
07
08     @GET
09     @Path("/listar")
10     @Produces("application/json")
11     public List<ClienteModel> listar(){
12
13         try {
14             String query = "select c from ClienteModel c";
15             List<ClienteModel> clientes = objEM.createQuery(query, ClienteModel.class).getResultList();
16             objEM.close();
17             return clientes;
18         } catch (Exception e) {
19             throw new WebApplicationException(500);
20         }
21     }
22
23     @GET
24     @Path("/buscar/{id_cliente}")
25     @Produces("application/json")
26     public ClienteModel buscar(@PathParam("id_cliente") int id_cliente){
27         try {
28             ClienteModel cliente = objEM.find(ClienteModel.class, id_cliente);
29             objEM.close();
30             return cliente;
31         } catch (Exception e) {
32             throw new WebApplicationException(500);
33         }
34     }
35
36     @POST
37     @Path("/cadastrar")
38     @Consumes("application/json")
39     public Response cadastrar(ClienteModel objClinte){
40         try {
41             objEM.getTransaction().begin();
42             objEM.persist(objClinte);
43             objEM.getTransaction().commit();
44             objEM.close();
45             return Response.status(200).entity("cadastro realizado.").build();
46         } catch (Exception e) {
47             throw new WebApplicationException(500);
48         }
49     }
50
51     @PUT
52     @Path("/alterar")
53     @Consumes("application/json")
54     public Response alterar(ClienteModel objClinte){
55         try {
56             objEM.getTransaction().begin();
57             objEM.merge(objClinte);
58             objEM.getTransaction().commit();
59             objEM.close();
60             return Response.status(200).entity("cadastro alterado.").build();
61         } catch (Exception e) {
62             throw new WebApplicationException(500);
63         }
64     }
65 }
```

```
64 }
65
66 @DELETE
67 @Path("/excluir/{id_cliente}")
68 public Response excluir(@PathParam("id_cliente") int id_cliente){
69     try {
70         ClienteModel objClinte = objEM.find(ClienteModel.class, id_cliente);
71
72         objEM.getTransaction().begin();
73         objEM.remove(objClinte);
74         objEM.getTransaction().commit();
75         objEM.close();
76
77         return Response.status(200).entity("cadastro excluído.").build();
78     } catch (Exception e) {
79         throw new WebApplicationException(500);
80     }
81 }
82 }
```

Neste código podemos verificar que os verbos do protocolo HTTP (GET, POST, PUT e DELETE) são utilizados para cada operação do CRUD. O verbo POST, por exemplo, é empregado para a operação relacionada ao cadastro de clientes. Para alteração dos dados foi adotado o PUT, DELETE para exclusão e GET para obter a listagem dos clientes. No código ainda podemos verificar algumas anotações do JAX-RS e o uso de recursos da JPA.

A seguir são apresentados os primeiros detalhes da classe **ClienteService**, que será analisada mais a fundo nos próximos tópicos:

- **Linha 03:** Local onde definimos que a classe **ClienteService** será um serviço. Para isso foi inserida a anotação **@Path** passando como parâmetro o nome do serviço que irá compor a URL de acesso aos recursos do servidor.

A partir disso, o caminho para acesso ao serviço será: *http://localhost:8080/apirest/cliente/*. Lembre-se que *apirest* e a porta foram definidos no mapeamento do servlet do Jersey no *web.xml*;

- **Linhas 05 e 06:** Na linha 5 foi criado um objeto da classe **JpaEntityManager**. Logo depois, na linha 6 é criado outro objeto, chamado **objEM**. Este recebe um **EntityManager** através da chamada ao método **getEntityManager()** de **JpaEntityManager**. É o **objEM** que irá possibilitar o acesso a dados.

Para melhor entendimento e mostrar na prática o consumo dos métodos (recursos) do nosso serviço de cadastro de cliente, iremos instalar um complemento do Firefox chamado de *HttpRequester*. Com esse intuito, acesse a opção *Complementos* deste navegador, procure por "HttpRequester" e então clique em *Instalar*. Este possibilita realizar requisições a serviços REST utilizando vários verbos do protocolo HTTP.

## Consumo do recurso listar clientes

O primeiro recurso a ser implementado será o de listagem dos dados. Desta forma, começaremos analisando o código relacionado à listagem (vide método **listar()** da **Listagem 7**) e como consumir este serviço.

A implementação deste método, equivalente à operação Read do CRUD, é explicada a seguir:



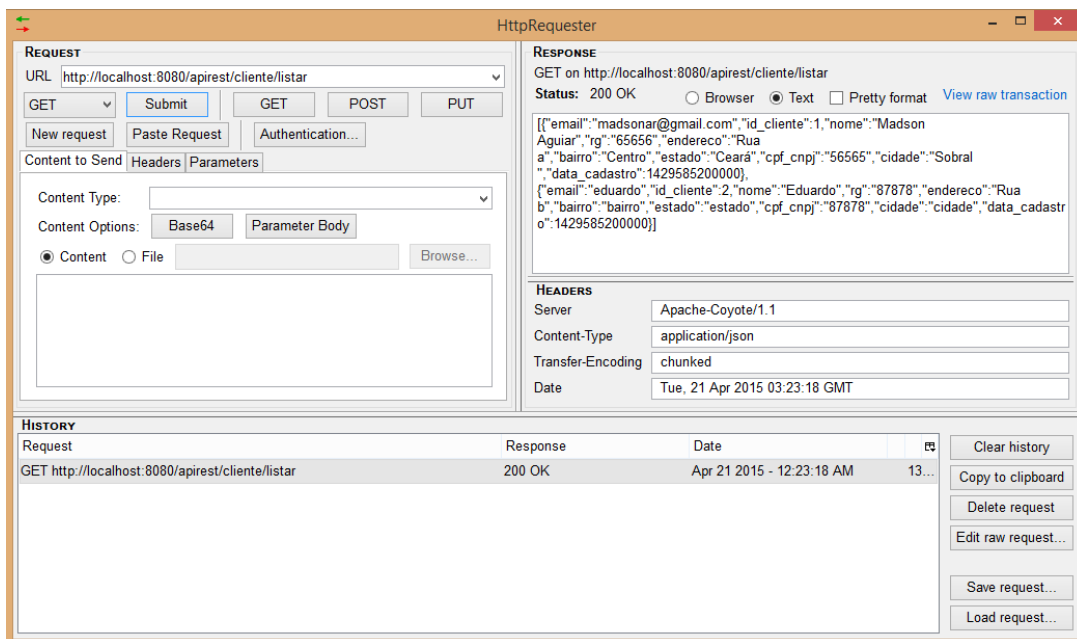
· **Linhas 08 a 21:** Este bloco de código define o primeiro método do serviço, o **listar()**, que retorna uma lista de **ClienteModel**. Na linha 08 foi utilizada a anotação **@GET** para informar que este método só poderá ser requisitado por uma requisição do tipo GET no protocolo HTTP.

Na linha 09 é informada a anotação **@Path**, que recebe por parâmetro o nome do recurso do serviço; neste caso o nome final da URL que aponta para o recurso de listagem de clientes no servidor, *listar*. Vale ressaltar que o valor da propriedade **name** de **@Path** não precisa ser o mesmo nome do método iniciado na linha 11, mas deixamos o mesmo apenas para facilitar o entendimento.

Na linha 10 foi especificada a última anotação do método listar: **@Produces**. Esta serve para informar ao Jersey que ele deve retornar ao cliente a listagem da linha 17 no formato JSON;

· **Linha 13 a 20:** Esta é a implementação do código que efetivamente irá buscar a listagem de clientes, onde é definida uma **String** chamada **query** que recebe a consulta a ser realizada no banco de dados. Em seguida, na linha 15 o objeto **clientes** recebe o resultado da consulta através da chamada aos métodos **createQuery()** e **getResultList()**, disponíveis no objeto **EntityManager**, e na linha 16 é encerrado o **EntityManager** chamando o método **close()**. Por fim, na linha 19 é criado um **throw** através da classe **WebApplicationException** para recuperarmos possíveis erros durante a execução e informar ao solicitante do serviço.

Na **Figura 13** expomos o resultado de uma requisição do tipo GET ao recurso listar clientes. Para isso, perceba que precisamos apenas informar a URL do serviço e selecionar o verbo GET. Antes de realizar o teste, no entanto, abra algum gerenciador de banco de dados MySQL e insira alguns clientes na tabela *cliente*.



[abrir imagem em nova janela](#)

**Figura 13.** Consumo do recurso listar clientes.

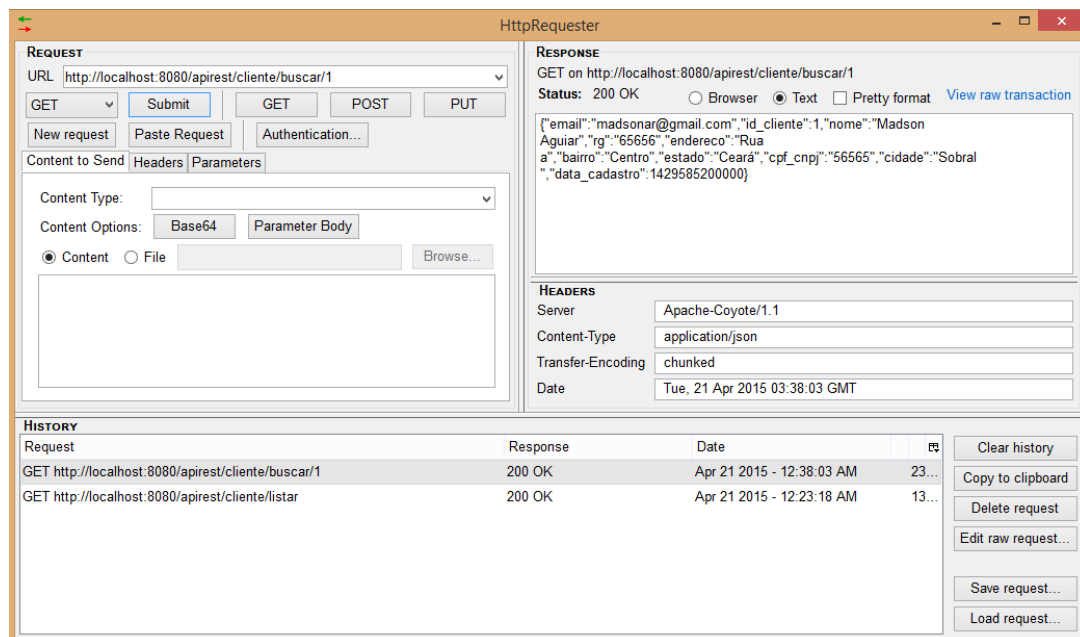
## Consumo do recurso buscar cliente

Outro importante recurso do nosso serviço é a busca de clientes na base de dados pelo id. Relacionado à operação Read do CRUD, a requisição a este recurso geralmente é realizada através de uma requisição do tipo GET.

No código do projeto exemplo, o método **buscar()** recebe o **id** do cliente a ser pesquisado e retorna os dados do mesmo no formato JSON, quando encontrado. Façamos uma análise do seu código:

- **Linhas 23 a 26:** Nestas linhas são declaradas as anotações **@GET**, **@Path** e **@Produces** para indicar, respectivamente, o tipo de método HTTP, o caminho de acesso ao recurso e o tipo dos dados que serão retornados (JSON, neste caso). Perceba ainda que na linha 24 é informado um parâmetro chamado **id\_cliente** para receber o código do cliente na requisição GET. É através deste parâmetro que é obtido o código a ser pesquisado no banco de dados;
- **Linha 26:** Aqui temos a declaração do método **buscar()**, local onde também fazemos uso da anotação **@PathParam**, que recebe o nome do parâmetro a ser passado com o código do cliente junto à URL de requisição;
- **Linha 28:** Nesta linha é criado um objeto do tipo **ClienteModel** para receber os dados do cliente pesquisado na base de dados. Veja que a consulta foi realizada através do método **find()**, que recebeu o tipo do model a ser pesquisado e o id do cliente. Nas linhas 29 e 30 o **EntityManager** é encerrado e é retornado o cliente pesquisado no formato JSON.

Na **Figura 14** podemos verificar o resultado do consumo do recurso buscar cliente através de uma requisição do tipo GET com a ferramenta Open HttpRequester.



### [abrir imagem em nova janela](#)

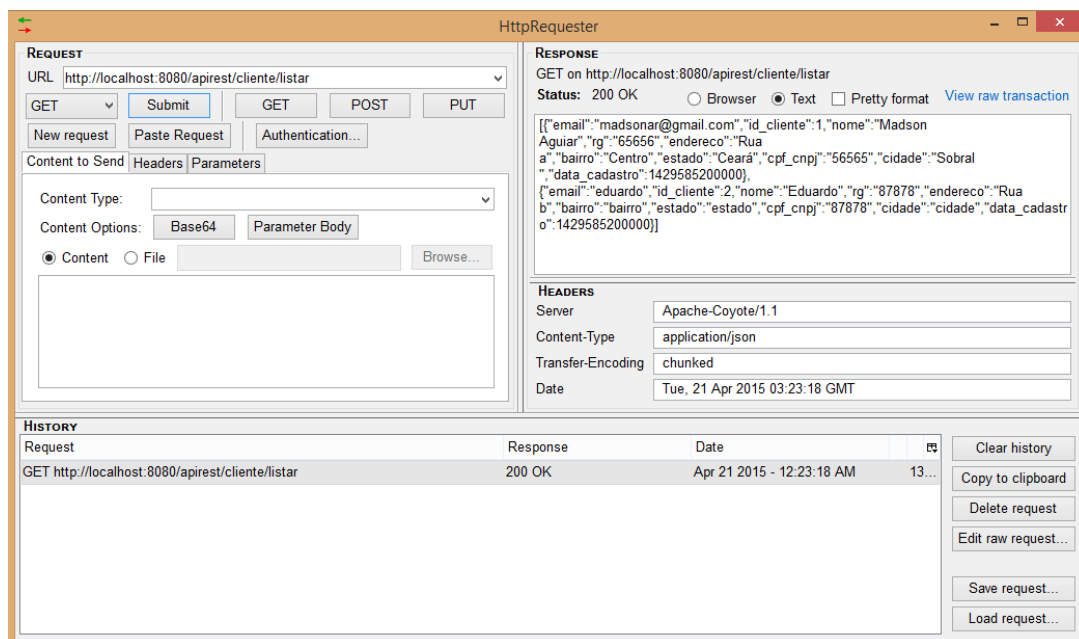
**Figura 14.** Consumo do recurso buscar cliente.

## Consumo do recurso cadastrar cliente

A próxima operação do CRUD que codificamos como um serviço é o Create, para cadastrar um novo cliente na base de dados. Para isso, devemos realizar uma requisição do tipo POST passando no corpo da mesma os dados do novo cliente. Os detalhes de sua implementação são apresentados a seguir:

- **Linha 38:** O primeiro ponto a destacar neste código é a presença de uma nova anotação, chamada **@Consumes**. Esta tem como função definir o tipo de dado a ser recebido pela requisição POST, neste caso o JSON;
- **Linha 39:** Nesta linha temos a declaração do método, que como parâmetro recebe o objeto cliente a ser cadastrado no banco;
- **Linhas 41 a 43:** Neste intervalo é iniciada uma nova transação com o banco de dados. Na linha 42 um novo cliente é persistido. Por fim, é realizado um commit para que o mesmo seja gravado permanentemente;
- **Linha 46:** Nesta linha é utilizado o método **status()** da classe **Response** para retornar ao cliente o código 200 do HTTP. Isto informa que o cliente foi cadastrado com sucesso.

Na **Figura 15** podemos verificar o resultado da requisição POST ao recurso cadastrar. Perceba que os dados do novo cliente estão no formato JSON, pois este é o formato esperado pelo servidor, conforme explicitado pela anotação **@Consumes("application/json")**.



[abrir imagem em nova janela](#)

**Figura 15.** Consumo do recurso cadastrar cliente.

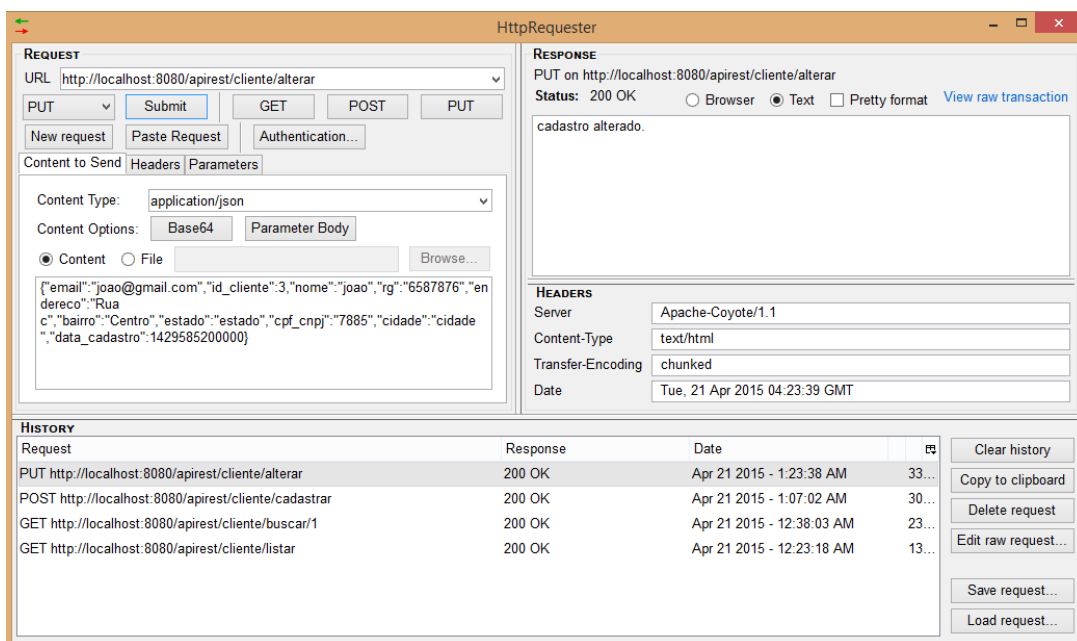
## Consumo do recurso alterar cliente

Outra operação do nosso serviço, relacionada ao Update do CRUD, é a alteração dos dados de um cliente. Como esperado, o método relacionado altera os dados do cliente e depois retorna uma mensagem informando o sucesso da execução. Analisemos o seu código:

- **Linha 51:** Nesta linha é informada a anotação **@PUT**, definindo que a requisição deve usar o método PUT do protocolo HTTP;

- **Linha 53:** Aqui é especificada a anotação **@Consumes** para informar que os dados recebidos pela requisição devem estar no formato JSON. Deste modo o Jersey poderá fazer a conversão para objetos Java;
- **Linhas 56 a 58:** Neste intervalo é iniciada uma nova transação (vide linha 56). Em seguida os dados do cliente são alterados na base através da chamada ao método **merge()** do **EntityManager**, que recebe por parâmetro o cliente a ser atualizado. Por fim, na linha 58 é realizado o commit para persistir essa mudança;
- **Linha 60:** Nesta linha é utilizada novamente a classe **Response**, que através do método **status()** retorna o código 200 informando que o cadastro do cliente foi alterado com sucesso.

Na **Figura 16** é possível verificar o consumo do recurso que altera as informações de um cliente no banco de dados.



[abrir imagem em nova janela](#)

**Figura 16.** Consumo do recurso alterar cliente.

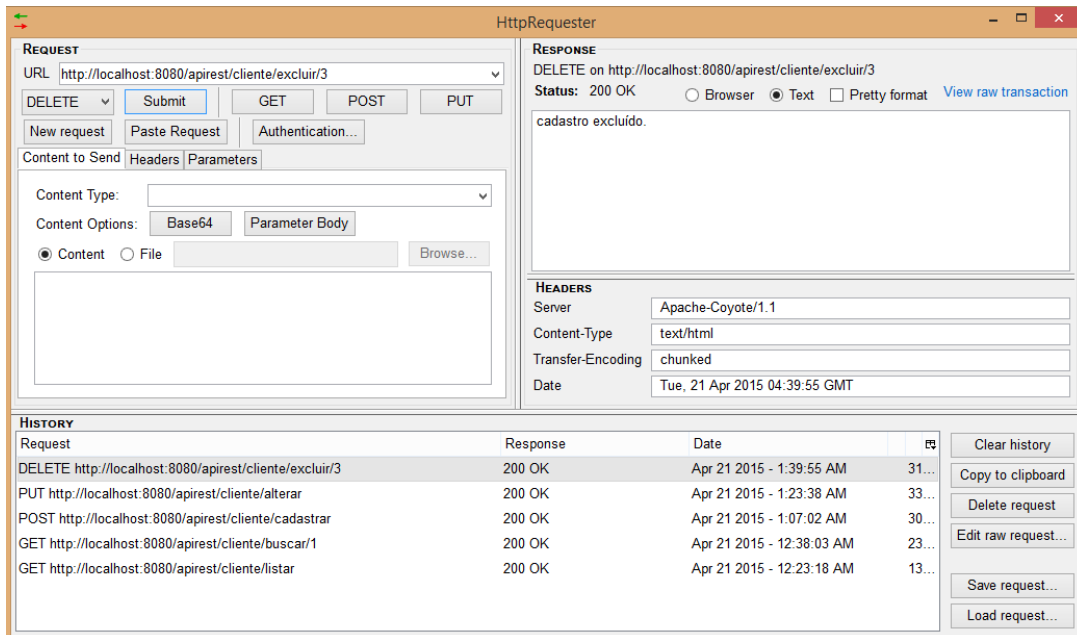
## Consumo do recurso excluir cliente

A última operação a ser implementada é a de excluir clientes, referente ao Delete do CRUD. Para isso, este método recebe o ID do cliente cadastrado na base de dados e então o exclui. Vejamos a sua implementação:

- **Linha 66:** Aqui foi utilizada a anotação **@DELETE**, que define que somente requisições com o verbo DELETE – do protocolo HTTP – podem requisitar este recurso do serviço;
- **Linha 67:** Perceba nesta linha que é informado como parâmetro o id do cliente a ser excluído da base de dados. Para que o serviço possa receber o parâmetro é necessário informar o ID do cliente no final da URI;
- **Linha 70:** Nesta linha podemos verificar a busca no bando de dados do cliente a ser excluído;

- **Linhas 72 a 74:** Neste intervalo declaramos uma transação que exclui o cliente do banco de dados através da chamada ao método **remove()** de **EntityManager**. Como parâmetro este método recebe o cliente. Na linha 74 efetuamos o **commit()**;
- **Linha 77:** Por fim, é retornado o código 200 para indicar que o cliente foi excluído com sucesso.

Veja na **Figura 17** o resultado do consumo do recurso relacionado à exclusão de um cliente no banco de dados.



### **abrir imagem em nova janela**

**Figura 17.** Consumo do recurso excluir cliente.

O desenvolvimento de aplicações que adotam o padrão arquitetural REST segue em alta e provavelmente continuará assim até que uma solução mais simples e leve seja apresentada à comunidade. Deste modo, se você ainda não está utilizando esta solução em suas aplicações que requerem web services, este artigo serve como um ótimo ponto de partida para isso.

Com a necessidade de soluções cada vez mais integradas, o uso de serviços já se tornou uma forte vertente no mundo do desenvolvimento. Conhecer e saber como implementá-los é, portanto, um requisito essencial a todo profissional que atua na área.

#### **Links**

##### **Endereço para download do Eclipse Luna.**

<https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunars2>

##### **Página do projeto DBDesigner Fork.**

<http://sourceforge.net/projects/dbdesigner-fork/>

##### **Dissertação de Roy Fielding.**

[https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\\_dissertation.pdf](https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf)

**Página do projeto Jersey.**

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

**Endereço para download do MySQL.**

<http://www.mysql.com/downloads/>

**Java Persistence API.**

<http://www.oracle.com/technetwork/java/javasee/tech/persistence-jsp-140049.html>

**What are RESTful Web Services?**

<https://docs.oracle.com/javaee/6/tutorial/doc/gjqy.html>



**Madson Aguiar Rodrigues**

Formação acadêmica em Análise e Desenvolvimento de Sistemas pela UNOPAR, pós-graduação em Engenharia de Sistemas pela ESAB e especialista em Tecnologias para aplicações Web pela UNOPAR. Trabalha com desenvolvimento de software há [...]

*Publicado em 1899*