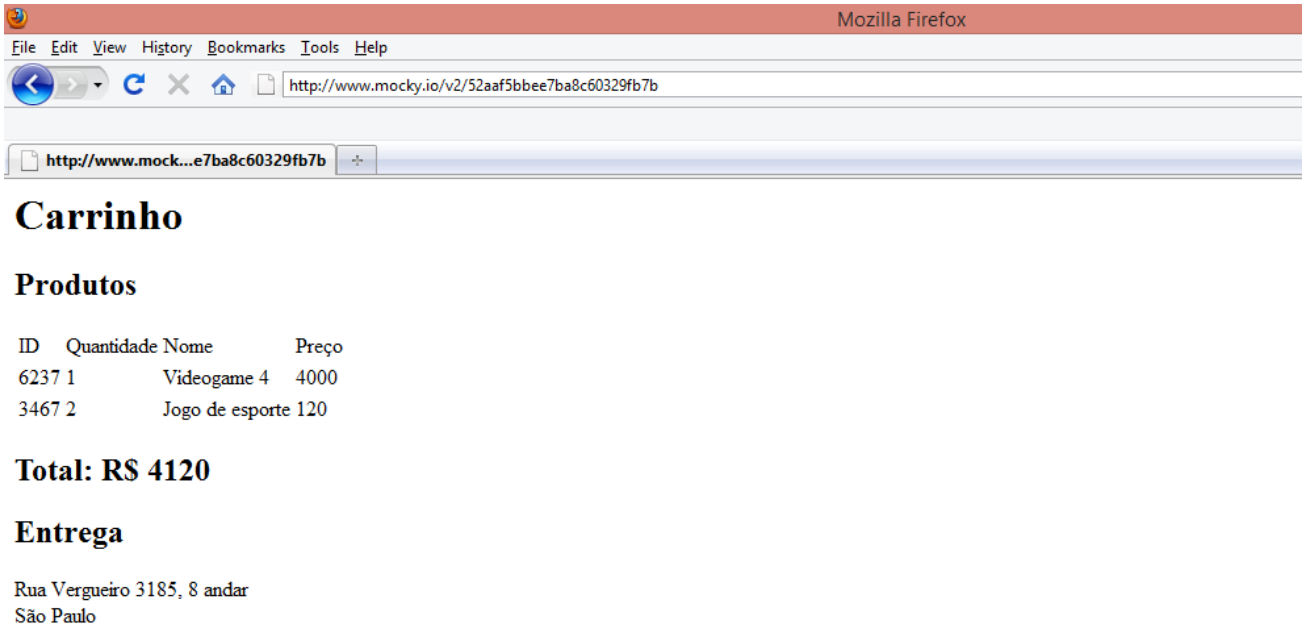
02

Serviços Web REST e Addressability

Bem vindo ao nosso curso de Web Services Rest com JAX-RS e Jersey. O que veremos neste curso é como fazer dois programas distintos se comunicarem na Internet. Acontece que hoje em dia é muito comum que nosso sistema esteja na internet e não viva isolado: ele quer se comunicar com outros sistemas, sistemas que foram escritos por outros desenvolvedores, em outras empresas, em outros lugares do mundo, em outras linguagens de programação.

Quero escrever um código que possa se comunicar com todos esses sistemas. Para fazer isso, uma das soluções que existe no mercado é o que chamamos de serviços baseados na web, os web services. Veremos nesse curso os web services que chamamos de REST.

Para começar imagine que temos um sistema que suporta a compra de produtos através de um carrinho: ele suporta adicionar produtos no carrinho, remover produtos etc. Provavelmente eu tenho uma página que me mostra o carrinho que estou comprando. Vou acessar aqui uma página que já mostra para mim um carrinho de compra. Acessamos o link http://www.mocky.io/v2/52aaf5bbee7ba8c60329fb7b (http://www.mocky.io/v2/52aaf5bbee7ba8c60329fb7b) e temos um carrinho de compra:



Já conhecemos um carrinho de compra HTML. Como sei que é HTML? Clico da direita e escolho *View page source*, o resultado sendo:

```
<html>
<head>
 <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
</head>
    <h1>Carrinho</h1>
    <h2>Produtos</h2>
    <table>
<tr><td>ID</td><td>Quantidade</td><td>Nome</td><td>Preço</td>
<tr>
<td>
6237</td>
```

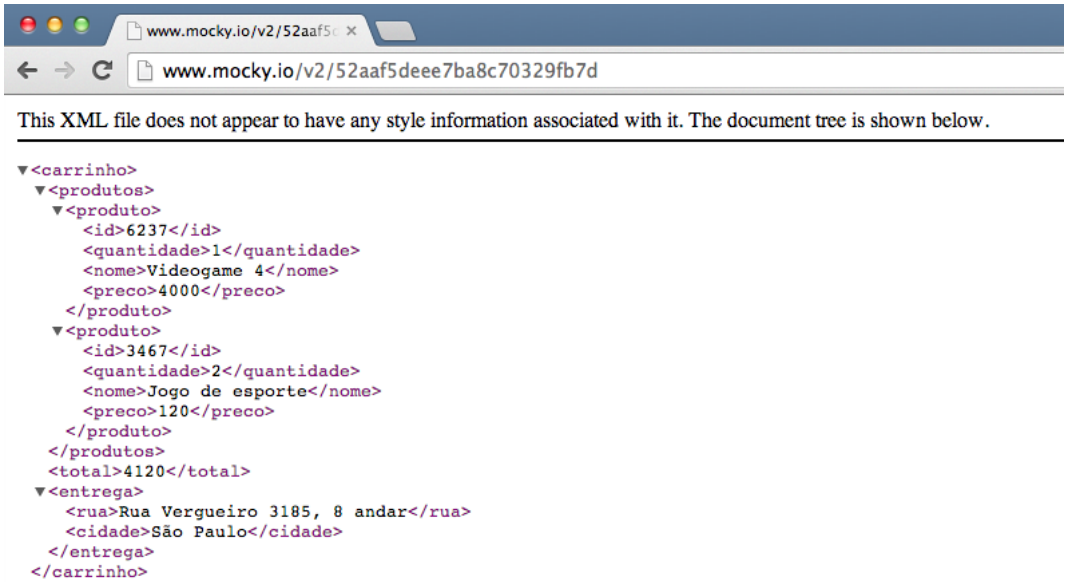
```
            <td>1</td>
            <td>Videogame 4</td>
            <td>4000</td>
        </tr>
        <tr>
            <td>3467</td>
            <td>2</td>
            <td>Jogo de esporte</td>
            <td>120</td>
        </tr>
</table>
    <h2>Total: R$ 4120</h2>
    <h2>Entrega</h2>
        Rua Vergueiro 3185, 8 andar<br/>
        São Paulo
</html>
```

Então quando faço uma requisição para o servidor, usando o link acima, o programa navegador (browser, o cliente) faz uma requisição para um servidor (outro programa), que nos devolve uma representação do carrinho. Cada página, cada link, ou melhor, cada URI que eu acesso representa um carrinho diferente contido no meu servidor. Cada carrinho que eu tenho tem uma URI diferente. Óbvio, o carrinho em si não é este HTML, o carrinho em si não é a URI, ele é algo que está no servidor, provavelmente os dados que estão no banco de dados do servidor. O HTML que é enviado de um lado para o outro, do servidor para o cliente, é uma representação do carrinho - uma *representation*.

Quando eu acesso a URI que identifica o carrinho, ele vem com a representação do carrinho para mim.

Mas vou ficar escrevendo programas que leem HTML e entender a tabela? Qual campo é qual campo? Vou ter que fazer o *scrapping* do HTML para entender os dados do carrinho? Trabalhoso né? As vezes o servidor é um programa antigo que só suporta HTML e temos que escrever clientes que fazem esse *scrap*. Mas na prática, hoje em dia, os servidores já se preparam disponibilizando o dado em outro formato, e não HTML. Algum outro formato, mais fácil de nossa máquina entender.

Vou acessar agora uma outra URI, http://www.mocky.io/v2/52aaf5deee7ba8c70329fb7d (http://www.mocky.io/v2/52aaf5deee7ba8c70329fb7d) uma URI que representa outro carrinho, e repara que o resultado retornado é uma representação em XML.



Os dados da representação são então uma representação do nosso carrinho, e temos lá dois produtos diferentes, com quantidades diferentes. Tenho um XML enviado de um lado para o outro, temos uma representação feita via XML. E tudo isso é muito importante quando falamos de REST.

Temos uma URI que identifica o nosso carrinho, o nosso *recurso* que está no servidor. E através dessa URI recebo a *representação* de nosso carrinho, que poderia ser JSON, XML, HTML, tem diversos formatos - ou mais especificamente diferentes media types, que podemos trabalhar.

Cada URI representa um carrinho, um recurso diferente, seja ele um produto, usuário, boleto, compra etc. Vamos então agora fazer um primeiro cliente que acessa essa URI, acessa essa representação, e verifica os dados que foram retornados? Para fazer isso vamos baixar os jars do JAX-RS e de sua implementação, o Jersey. Já temos um projeto configurado com esse JAR, baixe o arquivo zip a seguir: https://github.com/alura-cursos/webservices-rest-com-jaxrs-e-jersey/raw/master/loja.zip (https://github.com/alura-cursos/webservices-rest-com-jaxrs-e-jersey/raw/master/loja.zip) ou faça um fork e clone do projeto do github https://github.com/alura-cursos/webservices-rest-com-jaxrs-e-jersey (https://github.com/alura-cursos/webservices-rest-com-jaxrs-e-jersey)

Descompacte o arquivo e temos o diretório chamado loja. Vamos importar o projeto dentro do Eclipse. Vamos no Eclipse e escolhemos 'File, Import', 'Existing Projects into Workspace', escolhemos o root directory como sendo o diretório 'loja' que foi descompactado. Ele detecta o projeto e damos 'Finish'. Perfeito, o projeto é importado.

O que tenho aqui dentro do projeto? Tenho o diretório *src/main/java* com meu código java e o *src/test/java* para colocar futuros testes que escreveremos daqui a pouco. A primeira coisa que queremos fazer é criar um cliente, vamos então criar um teste que testa acessar essa URI e verifica que o XML resultante é o que esperamos, isto é, faz a requisição HTTP e confere que a representação que o servidor devolve é o que esperamos. Vamos lá? 'File, new, new class', o nome da classe será 'ClienteTest' (em inglês a palavra Test), no nosso pacote *br.com.alura.loja*. Dentro dessa classe colocaremos o primeiro método de teste:

```
public void testaQueAConexaoComOServidorFunciona() {

}
```

Colocamos uma anotação de teste para dizer que isto é um teste:

```
public class ClienteTest {

    @Test
    public void testaQueAConexaoComOServidorFunciona() {

    }

}
```

Uso o CTRL+SHIFT+O para importar a anotação. Dentro desse código de teste queremos um cliente http para acessar o servidor, portanto criamos um cliente novo:

```
Client client = ClientBuilder.newClient();
```

Ao importar novamente com CTRL+SHIFT+O lembre-se de escolher a classe Client do pacote *javax.ws*. Agora que temos um cliente, queremos usar uma URI base,a URI do servidor, para fazer várias requisições. No nosso caso é a URI do

servidor que estamos utilizando, o *www.mocky.io (http://www.mocky.io)*, portanto dizemos ao nosso cliente que trabalharemos com o alvo *http://www.mocky.io (http://www.mocky.io)*:

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://www.mocky.io");
```

Legal, vamos agora dizer que queremos fazer uma requisição para uma URI específica, target, por favor, para esse path '/v2/52aaf5deee7ba8c70329fb7d' faça uma requisição, e a requisição que faremos é a mais básica, a que pega dados do servidor, o método get:

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://www.mocky.io");
target.path("/v2/52aaf5deee7ba8c70329fb7d").request().get();
```

Mas se fazemos uma requisição get, estamos interessado nos dados que foram devolvidos pelo servidor, portanto ele devolve uma resposta, mas dessa vez estamos interessados somente na String que ele nos devolve, portanto passamos uma *String.class* para que ao executar o método get e receber os resultados, ele converta o corpo da resposta em uma String, algo bem simples, e devolva essa String para nós:

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://www.mocky.io");
String conteudo = target.path("/v2/52aaf5deee7ba8c70329fb7d").request().get(String.class);
```

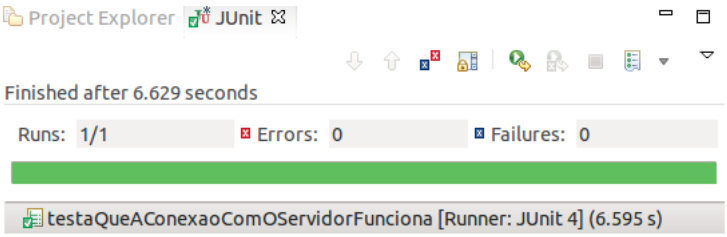
Após fazer a requisição, o cliente devolve o conteúdo para nós. Queremos agora ter certeza que o conteúdo contem a 'Rua Vergueiro 3185', que ela contem o pedaço do XML que nos interessa. Nesse caso estou dizendo que somente estou interessado na rua:

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://www.mocky.io");
String conteudo = target.path("/v2/52aaf5deee7ba8c70329fb7d").request().get(String.class);
Assert.assertTrue(conteudo.contains("Rua Vergueiro 3185"));
```

Como esse pedaço do xml começa com a tag deixarei isto bem claro em nosso assert:

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://www.mocky.io");
String conteudo = target.path("/v2/52aaf5deee7ba8c70329fb7d").request().get(String.class);
Assert.assertTrue(conteudo.contains("<rua>Rua Vergueiro 3185"));
```

Só estou verificando que um pedaço do xml está lá dentro, qual o motivo? Pois só estou interesado em saber se a conexão com o servidor está funcionando. Rodamos o teste clicando da direita, 'Run as', 'JUnit test', e o resultado é verde!



Claro, meu cliente poderia executar o que eu tivesse interesse, verificando todo o xml ou qualquer coisa do genero, mas nosso teste por enquanto se baseia em garantir que somos capazes de, como clientes, acessar o servidor e extrair as informações de um recurso. E ele passa! Só para garantir que o XML retornado era o que esperava, vamos rodar uma vez um Sysout:

```
@Test
public void testaQueAConexaoComOServidorFunciona() {
    Client client = ClientBuilder.newClient();
    WebTarget target = client.target("http://www.mocky.io");
    String conteudo = target.path("/v2/52aaf5deee7ba8c70329fb7d").request().get(String.class);
    System.out.println(conteudo);
    Assert.assertTrue(conteudo.contains("Rua Vergueiro 3185"));
}
```

E o resultado no console é o que esperávamos:

```
<carrinho>
    <produtos>
        <produto>
            <id>6237</id>
            <quantidade>1</quantidade>
            <nome>Videogame 4</nome>
            <preco>4000</preco>
        </produto>
        <produto>
            <id>3467</id>
            <quantidade>2</quantidade>
            <nome>Jogo de esporte</nome>
            <preco>120</preco>
        </produto>
    </produtos>
    <total>4120</total>
    <entrega>
        <rua>Rua Vergueiro 3185, 8 andar</rua>
        <cidade>São Paulo</cidade>
    </entrega>
</carrinho>
```

Removemos o Sysout e continuamos. O cliente funciona como esperávamos, ele acessa um servidor do outro lado do mundo e extrai as informações que queremos. Mas nesse curso queremos escrever não só clientes e testes automatizados, queremos escrever também o código do servidor. Então vamos escrever nosso servidor! Primeiro escrevemos o servidor em si.

Vamos olhar nossos pacotes? Dentro do pacote de modelo temos a classe Carrinho, que tem uma lista de produtos, a rua, a cidade e um id. Já o Produto tem o preço, o id, o nome e a quantidade. Temos essas duas classes, bonitas, mas quero fazer agora uma classe que representa um carrinho na internet, um recurso que vai devolver o XML de um carrinho. Eu já tenho o modelo, tenho um DAO que acessa o banco de dados em memória (você pode usar outra biblioteca para acessar seu banco, sinta-se a vontade quando escrever seu próprio projeto, claro). Com o CarrinhoDAO busco o carrinho, e já temos o carrinho de ID 1, que tem alguns produtos. Quero criar agora a classe que representa o recurso, que busca os dados no banco de dados (usando o DAO) e transforma em XML para mostrar ao cliente. Quero criar um recurso web. Vou criar no pacote `resource` uma classe chamada `CarrinhoResource`. Todo `Resource` do JAX-RS será anotado com a anotação `Path`, para dizer qual a URI que acessará esses recursos, no nosso caso queremos 'http://servidor/carrinhos' (http://servidor/carrinhos'), isto é '/carrinhos', portanto escrevemos somente 'carrinhos':

```
@Path("carrinhos")
public class CarrinhoResource {

}
```

Vou colocar meu método que busca carrinhos:

```
public void busca() {
}
```

E o que ele vai fazer? Buscar o carrinho de ID 1:

```
public void busca() {
    Carrinho carrinho = new CarrinhoDAO().busca(1l);

}
```

E o que vamos retornar? Queremos devolver uma representação deste carrinho em XML, portanto retornamos o carrinho em sua versão XML:

```
public void busca() {
    Carrinho carrinho = new CarrinhoDAO().busca(1l);
    return carrinho.toXML();
}
```

Então tanto no cliente podemos trabalhar com a String de XML em si, aqui podemos também devolver em nosso método uma String XML, que será retornada para o cliente. Sinta-se a vontade. Para serializar nosso objeto em XML você pode usar qualquer biblioteca que achar interessante, no nosso caso usaremos primeiro nossa própria biblioteca. Primeiro alteramos o retorno do método para String:

```
public String busca() {
    Carrinho carrinho = new CarrinhoDAO().busca(1l);
```

```
    return carrinho.toXML();
}
```

Mas já que o método devolve uma String, como o servidor será capaz de dizer ao cliente que o que está sendo devolvido, aquilo que está sendo produzido, é um XML e não um JSON, por exemplo? O que o método produz? Devemos dizer que o que será produzido é um XML:

```
@Produces(MediaType.APPLICATION_XML)
public String busca() {
    Carrinho carrinho = new CarrinhoDAO().busca(1l);
    return carrinho.toXML();
}
```

Falamos que o método gera APPLICATION_XML, mas também devemos falar que ele será acessado via *GET*, o método tradicional para buscar informações do servidor para o cliente:

```
@GET
@Produces(MediaType.APPLICATION_XML)
public String busca() {
    Carrinho carrinho = new CarrinhoDAO().busca(1l);
    return carrinho.toXML();
}
```

Agora estamos prontos para criar o método `toXML` em nosso `Carrinho`:

```
public String toXML() {
    return null;
}
```

Escrevemos aí o que desejamos, usamos uma biblioteca maluca, o XStream, o JAXB, concatenar String, o que você quiser! Para mostrar que podemos usar qualquer biblioteca, nesse primeiro exemplo usaremos o XStream:

```
public String toXML() {
    return new XStream().toXML(this);
}
```

O XStream usará o padrão dele, mas podemos fazer aqui o que bem desejar. Se tiver interesse, você pode ver mais sobre como configurar o XStream no curso dele, já que aqui focaremos na parte de nossos webservices. Nosso código está pronto, agora basta levantar o servidor. Levante o servidor. Mas qual servidor? Onde está ele? Precisamos do servidor!

Como levantar um servidor com Jersey logo de cara? A maneira mais simples de todas é criar uma classe para nosso servidor, dentro do pacote `br.com.alura.loja` vou criar uma classe chamada Servidor. Ela terá o método `main`:

```
package br.com.alura.loja;

public class Servidor {

    public static void main(String[] args) {
```

```
    }

}
```

Dentro do método `main` quero levantar um servidor do Grizzly, compatível com JAX-RS, servlet api e muito mais:

```
HttpServer server = GrizzlyHttpServerFactory.createHttpServer();
```

Mas tenho que falar qual a uri e a porta que desejo abrir meu servidor, para isso passo como parâmetro essa URI:

```
package br.com.alura.loja;

import java.net.URI;

import org.glassfish.grizzly.http.server.HttpServer;
import org.glassfish.jersey.grizzly2.httpserver.GrizzlyHttpServerFactory;

public class Servidor {

    public static void main(String[] args) {
        URI uri = URI.create("http://localhost:8080/");
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(uri);
    }

}
```

Estamos rodando na porta 8080 pois é o padrão de servlet containers e não precisamos nos preocupar com permissões de admin para a porta 80. Criamos o servidor, mas como o Grizzly será capaz de entender que nossa aplicação é baseada em JAX-RS e não em servlet-api? Como ele saberá onde procurar nossa aplicação e o que utilizar? Precisamos passar uma configuração para o grizzly, vamos lá?

```
HttpServer server = GrizzlyHttpServerFactory.createHttpServer(uri, config);
```

Criamos então uma configuração:

```
ResourceConfig config = new ResourceConfig();
```

E falamos que desejamos buscar no pacote `br.com.alura.loja` , tudo que tem aí dentro, quero que você busque como JAX-RS e utilize como serviço:

```
ResourceConfig config = new ResourceConfig().packages("br.com.alura.loja");
```

Podemos agora imprimir uma mensagem que nosso servidor está no ar:

```
System.out.println("Servidor rodando");
```

E paramos o servidor quando o usuário apertar enter:

```
System.in.read();
server.stop();
```

Nosso código `main` final possui somente quatro linhas de código para levantar e parar o nosso servidor web, além de informações de Sysout:

```
public class Servidor {

    public static void main(String[] args) throws IOException {
        ResourceConfig config = new ResourceConfig().packages("br.com.alura.loja");
        URI uri = URI.create("http://localhost:8080/");
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(uri, config);
        System.out.println("Servidor rodando");
        System.in.read();
        server.stop();
    }

}
```

Claro, poderia extrair tudo isso e colocar em uma aplicação web, mas o Jersey permite a utilização ainda mais simples: basta levantar o Grizzly. Rodo a aplicação clicando da direita no servidor, `Run as` , `Java Application` . E agora vejo a mensagem no console de que o `Servidor rodando` . Vou no navegador e acesso `http://localhost:8080/carrinhos` . O resultado é o XML:

```
<br.com.alura.loja.modelo.Carrinho>
  <produtos>
    <br.com.alura.loja.modelo.Produto>
      <preco>4000.0</preco>
      <id>6237</id>
      <nome>Videogame 4</nome>
      <quantidade>1</quantidade>
    </br.com.alura.loja.modelo.Produto>
    <br.com.alura.loja.modelo.Produto>
      <preco>60.0</preco>
      <id>3467</id>
      <nome>Jogo de esporte</nome>
      <quantidade>2</quantidade>
    </br.com.alura.loja.modelo.Produto>
  </produtos>
  <rua>Rua Vergueiro 3185, 8 andar</rua>
  <cidade>S?o Paulo</cidade>
  <id>1</id>
</br.com.alura.loja.modelo.Carrinho>
```

E esse XML é exatamente o XML que o XStream retornou em nosso servidor. Lembrando que se você quiser configurar o XStream ou usar JAX-B ou qualquer outra biblioteca de outros media types e formatos, sinta-se a vontade para em produção alterar seu código.

Em aplicações modernas é comum que façamos a comunicação entre diversas aplicações, e a aplicação de hoje em dia costuma fazer isso através da web, mas não só através, ela usa a web, ela está na web, utilizando o protocolo HTTP. Os web services do REST servem para fazer isso de maneira mais inteligente, utilizando diversas vantagens do protocolo, e

a primeira das características que vimos é a Addressability, que todo recurso (por exemplo um carrinho) tem um endereço de identificação, uma URI, se eu entregar essa URI para o João ou para a Maria, ambos tem acesso ao mesmo carrinho. A URI identifica o recurso, e não o usuário que identifica o carrinho que está sendo acessado. Com o REST usamos a URI para identificar o recurso.

A segunda característica importante é que o XML ou qualquer outro dado sendo jogado de um lado para o outro não é o carrinho, não é o recurso, mas sim uma representação de nosso recurso.

E agora como eu testo para ver se está tudo funcionando? No próximo capítulo veremos como criar um teste REST, um teste que levanta seu servidor, executa o teste, garante que o resultado é o esperado, e então derruba o servidor: um teste end-to-end de um serviço rest. Agora vamos aos exercícios.