02

# Testes end-to-end com JAX-RS e Grizzly

Já temos o código do servidor que levanta o Grizzly em http://localhost:8080 (http://localhost:8080) e permite o acesso a URI /carrinhos que traz o nosso xml, a representação do carrinho 1. Podia ser um outro media type ou formato qualquer como JSON etc. Temos também uma classe de teste que testa a conexão com um servidor aleatório, de terceiros. Mas se este servidor cair ou for atualizado, não posso rodar meus testes. Quero rodá-los com o servidor que tenho utilizado, o Grizzly, e o JAX-RS. Vendo nosso teste:

```
@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);
    Assert.assertTrue(conteudo.contains("<rua>Rua Vergueiro 3185"));
}
```

Gostaríamos que ele não testasse mais o `http://www.mocky.io` mas sim o `http://localhost:8080` , e que o nosso servidor do Grizzly fosse rodado toda vez que o teste fosse executado! Queria também que o teste não fosse mais "testa que a conexão com o servidor funciona", mas sim que "testa que buscar um carrinho traz o carrinho esperado":

```
@Test
public void testaQueBuscarUmCarrinhoTrazOCarrinhoEsperado() {
    // codigo anterior
}
```

Continuamos criando o cliente normal, mas passamos a acessar o servidor `http://localhost:8080` e o path `/carrinhos` :

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://localhost:8080");
String conteudo = target.path("/carrinhos").request().get(String.class);
Assert.assertTrue(conteudo.contains("<rua>Rua Vergueiro 3185"));
```

Adicionamos novamente o Sysout só para verificar o conteudo recebido:

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://localhost:8080");
String conteudo = target.path("/carrinhos").request().get(String.class);
System.out.println(conteudo);
Assert.assertTrue(conteudo.contains("<rua>Rua Vergueiro 3185"));
```

Rodamos o servidor (importante não esquecer), e rodamos o teste. O xml resultante tem o conteúdo que queremos, a Rua Vergueiro 3185, mas mais bonito do que eu garantir somente a rua através desse contains, será eu deserializar esse XML de volta a um objeto do tipo `Carrinho` e ai garantir que esse carrinho é exatamente o que eu esperava. Removemos então a linha do assert, e usamos o XStream para deserializar:

```
Carrinho carrinho = (Carrinho) new XStream().fromXML(conteudo);
```

E com o carrinho em mãos, quero verificar que a rua será igual a vergueiro:

```
Carrinho carrinho = (Carrinho) new XStream().fromXML(conteudo);
Assert.assertEquals("Rua Vergueiro 3185, 8 andar", carrinho.getRua());
```

Legal, conferimos o teste e é um sucesso, o carrinho é deserializado e conferi que a rua dele é a rua esperada. Mas poxa, toda vez que quero rodar os testes tenho que levantar o servidor, rodar os testes, derrubar o servidor? Meio triste né? Já que o código para levantar o servidor é Java puro, podemos fazer isso dentro de nosso teste! Pegamos as três linhas que levantam o servidor e colocamos antes do teste, e a linha de `stop` logo após o teste:

```
public class ClienteTest {

    @Test
    public void testaQueBuscarUmCarrinhoTrazOCarrinhoEsperado() {
        ResourceConfig config = new ResourceConfig().packages("br.com.alura.loja");
        URI uri = URI.create("http://localhost:8080/");
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(uri, config);

        Client client = ClientBuilder.newClient();
        WebTarget target = client.target("http://localhost:8080");
        String conteudo = target.path("/carrinhos").request().get(String.class);
        Carrinho carrinho = (Carrinho) new XStream().fromXML(conteudo);
        Assert.assertEquals("Rua Vergueiro 3185, 8 andar", carrinho.getRua());

        server.stop();
    }

}
```

Antes de rodar, garantimos no console que o servidor parou, feche todos os consoles e programas que estavam rodando. Agora rodamos o teste: verde, sendo que o servidor levantou e foi derrubado no teste!

```
Apr 23, 2014 10:50:45 AM org.glassfish.jersey.server.ApplicationHandler initialize
INFO: Initiating Jersey application, version Jersey: 2.5 2013-12-18 14:27:29...
Apr 23, 2014 10:50:46 AM org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [localhost:8080]
Apr 23, 2014 10:50:46 AM org.glassfish.grizzly.http.server.HttpServer start
INFO: [HttpServer] Started.
Apr 23, 2014 10:50:46 AM org.glassfish.grizzly.http.server.NetworkListener stop
INFO: Stopped listener bound to [localhost:8080]
```

Tem alguns cuidados que devo tomar, claro, eu tenho que pegar o código inicial e executá-lo sempre antes de um teste, retiro o código e coloco no before:

```
@Before
public void startaServidor() {
    ResourceConfig config = new ResourceConfig().packages("br.com.alura.loja");
    URI uri = URI.create("http://localhost:8080/");
```

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

E preciso matar o servidor depois:

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

import java.net.URI;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;

import org.glassfish.grizzly.http.server.HttpServer;
import org.glassfish.jersey.grizzly2.httpserver.GrizzlyHttpServerFactory;
import org.glassfish.jersey.server.ResourceConfig;
import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import br.com.alura.loja.modelo.Carrinho;

import com.thoughtworks.xstream.XStream;

public class ClienteTest {

    private HttpServer server;

    @Before
    public void before() {
        ResourceConfig config = new ResourceConfig().packages("br.com.alura.loja");
        URI uri = URI.create("http://localhost:8080/");
        this.server = GrizzlyHttpServerFactory.createHttpServer(uri, config);
    }

    @After
    public void mataServidor() {
        server.stop();
    }

    @Test
    public void testaQueBuscarUmCarrinhoTrazOCarrinhoEsperado() {
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target("http://localhost:8080");
        String conteudo = target.path("/carrinhos").request().get(String.class);
        Carrinho carrinho = (Carrinho) new XStream().fromXML(conteudo);
        Assert.assertEquals("Rua Vergueiro 3185, 8 andar", carrinho.getRua());
    }

}
```

Só tem um problema, essas três linhas ainda estão repetidas: elas aparecem nessa classe e também na classe Servidor. Vou então nessa classe `Servidor`, seleciono as três linhas, clico da direita e escolho `Refactor, Extract Method` para

extrair um método com essas linhas, chamo o método de `inicializaServidor`, e o método é criado. Removo o `private`, colocando `public`:

```
public static HttpServer inicializaServidor() {
    ResourceConfig config = new ResourceConfig().packages("br.com.alura.loja");
    URI uri = URI.create("http://localhost:8080/");
    HttpServer server = GrizzlyHttpServerFactory.createHttpServer(uri, config);
    return server;
}
```

E dentro do nosso teste mudamos o método de before para chamar tal método:

```
@Before
public void before() {
    this.server = Servidor.inicializaServidor();
}
```

Agora não temos mais código repetido, rodamos o teste novamente e está tudo funcionando. O que vimos até agora? Já possuíamos o código de um servidor e o de um cliente que acessava um servidor qualquer na internet. Mas quando criamos código de teste end-to-end, queremos usar os métodos do tipo @Before ou @BeforeClass para levantar e derrubar o servidor, passando por um serviço total, e não por somente pedaços de nossa aplicação (como no caso de testes de unidade). Para saber mais sobre testes, não deixe de fazer os cursos de teste do Alura.

Agora que já somos capazes de criar testes end-to-end do nosso serviço, vamos para os exercícios e depois melhorar nosso serviço, aprendendo mais sobre REST.