Introdução prática a REST Web Services com Java usando RESTEasy

Publicado por jesuino em 27/03/2012 - 22.855 visualizações

Artigo Beta! Por favor, poste no fórum qualquer erro encontrado

Nesse presente artigo vamos falar rapidamente de REST no mundo Java usando a especificação JAX-RS e a implementação RESTEasy. Esse texto é ?mão na massa", não vamos a fundo aos príncipios REST e no funcionamento do JAX-RS. Se quiser saber mais você deve ler a tese de Roy Fielding ou ler os livros recomendados no final do capítulo.

Muito blá blá né? Vamos ao que interessa!

Ambiente

Você vai precisar de um ambiente de desenvolvimento comum a desenvolvedores Java:

- Ambiente Java 1.5+
- Eclipse Helius ou Indigo
- Algum servlet container, de preferência Tomcat 6
- Aproveite e já baixe o RESTEasy 2.3.2.

Olá Mundo!

Provavelmente você já sabe que tudo que vamos aprender no mundo da programação começamos com um ?Hello World" ou, no bom português, ?Olá Mundo!". O que vamos fazer agora é um mega simples Web Service (WS) que simplesmente diz ?Olá Mundo!". Pode parecer idiota, mas dele vamos aprender o básico de um REST WS usando Java.
Para construir WS baseados nos príncipios REST usamos alguma implementação da especificação JAX-RS. Essas especificação é uma das mais simples de ser lida e entendida e ela simplesmente vai te ajudar a direcionar requests HTTPs(lembre-se que com HTTP é possível seguir todos os príncipios REST) para suas classes POJO devidamente anotadas. A especificação também prevê diversos outros detalhes como parse do corpo da request HTTP diretamente para POJO entre muitos outros.
O que vamos fazer agora é simplesmente criar um Olá Mundo para servir de base para futuros projetos que usam JAX-RS e também entender o básico do básico de toda a especificação.
Conforme falamos algumas linhas acima, JAX-RS ajuda você a direcionar requests HTTP para suas classes. Para você fazer uma request você deve saber a URL de destino e o recurso que quer acessar dessa URL, a URI do recurso. JAX-RS permite, através de anotações Java, que você escolha a URI de cada um de seus recursos e o método HTTP desejado.
Resumo: Você cria um POJO(classes Java simples), anota, deploy e os métodos dessa classe irão responder à requests HTTP após o deploy em um servlet container.

Determinando o projeto

Antes de codificar vamos definir qual URI irá responder ?Olá Mundo!". Que tal ?ola-mundo"? Ok, vamos usar ola-mundo. E qual o método HTTP? GET. GET usamos para pegar coisas do servidor(PUT ou POST para criar, DELETE para apagar, veremos mais a frente...). Então, nosso primeiro WS REST responde como abaixo:

Método HTTP: GET
URI: /ola-mundo
Tipo de retorno: Texto Plano

Uma coisa nova que você deve ter notado é o tipo de retorno. REST WS respondem em qualquer formato de dado que você quiser: XML, JSON, Texto Plano, YAML, JPG, etc. Basta você ter o MessageBodyWriter para aquele tipo. Mas para nós isso não importa ainda, pois com texto plano não precisamos nos preocupar com isso.

Configurando

Vamos a parte mais legal que é programar. Tem muitas partes chatas que não vamos falar aqui, mas vamos dar link na medida do necessário. Assim evitamos entediar os sabidinhos, mas não deixamos os novatos na mão :-).
Uma aplicação JAX-RS é uma aplicação WEB simples que usa empacotamento WAR, logo podemos criar um projeto no Eclipse do tipo ?Dynamic Web Project":

- Abra o Eclipse, vá em File->New-> Dynamic Web Project
- Dê o nome ?OlaREST"(ou outro que você quiser) e configure a versão 2.5 no campo ?Dynamic Web Module Version";
- Mais detalhes na doc do Eclipse.

Esse seu projeto, por padrão, não reconhece nada de JAX-RS. Se tentar usar vai ter só erro e dor de cabeça e vai fechar o Eclipse e vai ir ver sua timeline do facebook ou o trending do 9gag :-(. Para evitar isso, siga os passos abaixo para adicionar o RESTEasy no ?buildpath" do seu projeto. Assim o Eclipse reconhecerá tudo que você usar de JAX-RS:

- Após baixar o RESTEasy, ?unzip" o arquivo baixado em algum lugar;
- Da pasta resultante de ter ?deszipado" o resteasy, vá pasta lib e copie todos os arquivos JAR ali e cole eles lá na pasta do seu projeto em

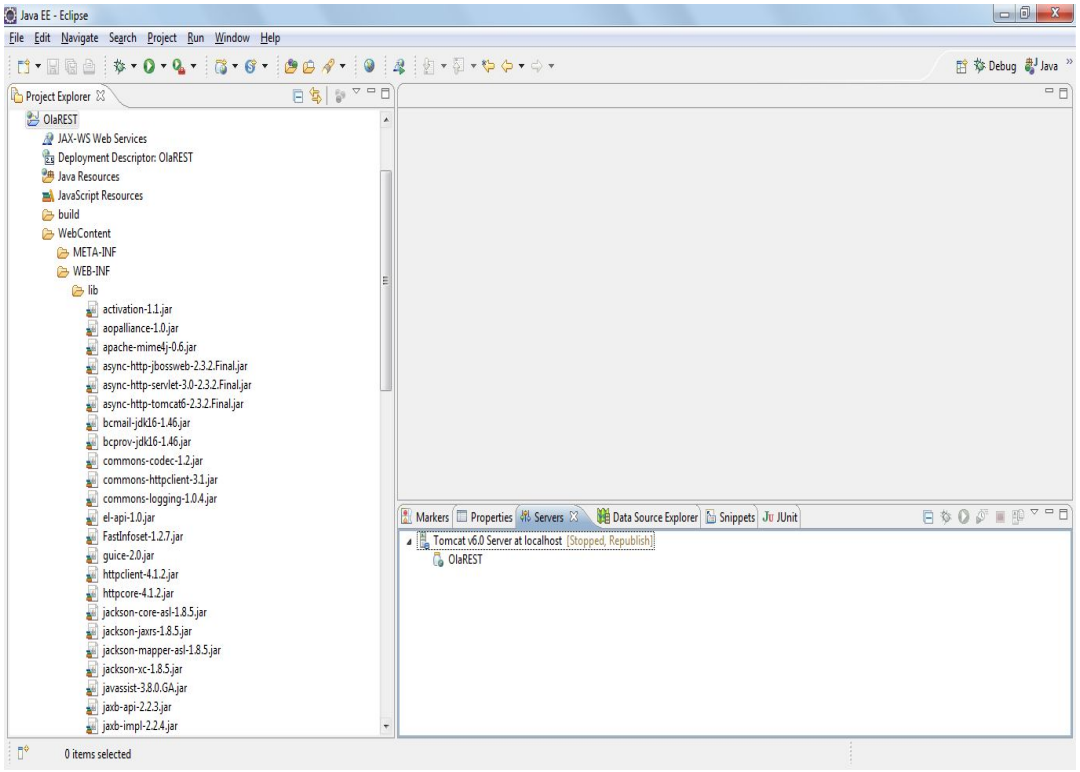
OlaREST/WebContent/WEB-INF/LIB. Você colar usando a própria interface do eclipse(pela IDE ir na pasta e apertar ?CTRL+V") ou colar na pasta pelo seu sistema operacional...
- No Eclipse, selecione os JARs da pasta lib do seu projeto (os que você acabou de colar), clique com o botão direito e selecione o menu ?Build Path" -> ?Add to Build Path". Agora o Eclipse irá reconhecer todas as classes relacionados ao RESTEasy.

Observe que colar todos os JAR no seu projeto WEB não é recomendado, você não vai usar todos nesse exemplo! As maiores desvantagens são maior tamanho em disco do projeto, possíveis conflitos de classloading quando for feito o deploy em servidores de aplicação, como o JBoss 5 e 6 e espalhamento desnecessário de diversos JARs por diversos projetos. Lembre-se disso quando for colocar aplicações em produção!

Estamos a um passo de deixar o ambiente no jeito para o desenvolvimento. Agora falta adicionar nosso servlet container ao Eclipse. O processo é melhor descrito nesse vídeo, mas abaixo vão os passos que segui na escrita desse texto:

- Baixei e descompactei o tomcat em uma pasta de minha preferência(ou você pode usar o .exe se preferir);
- No Eclipse acessei o menu File -> New -> Other -> selecionei Server e então cliquei em Next;
- No diálogo que apareceu eu selecionei Tomcat 6.x e cliquei em Next;
- Mostrei ao Eclipse onde estava o meu Tomcat através do campo ?Tomcat Installation Directory" e cliquei em Next;
- Selecionei o meu projeto ?OlaREST" no lado esquerdo ?Available" e passei para o lado direito ?Configured";

Se tudo deu certo até agora, o seu ambiente está mais ou menos assim:



Você pode, se quiser, iniciar o tomcat para testar ele. Para isso clique sobre ele na abinha que ele está e então no botão de play no lado superior direito das abas. Após ver ele inicializando acesse http://localhost:8080. Você deve pelo menos um 404 ou a página de boas vindas do servlet container mais famoso atualmente.

Pronto, agora é hora de programar!

Programando

A primeira coisa que vamos fazer é criar um pacote onde iremos colocar nossas classes de recursos. Vamos criar o pacote recursos! Para isso dentro do seu projeto clique com o botão direito em ?Java Resource/src" e selecione no meu New -> Package. Coloque o nome ?recursos" e então clique em ?finish".
Dentro desse pacote crie uma classe Java simples chamada ?RecursoOla".(botão direito sobre o pacote, new, class, nome ?RecursoOla" e então Finish ...). Abaixo está o código completo de nossa classe e vamos falar um pouco sobre ele nas próximas linhas:

```
package recursos;

import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("ola-mundo")
public class RecursoOla {

   @GET
   public String digaOla() {
      return "Olá Mundo!";
   }
}
```

A primeira coisa que devemos fazer é anotar nossa classe com @Path, assim ela será reconhecida como um ?recurso JAX-RS". Essa anotação também permite informar a qual URI esse recurso responde e no nosso caso ela irá responde a ?ola-mundo".
Perceba o método digaOla com a anotação @GET. Esse método será invocado quando uma request GET para a URI ?diga-ola" chegar ao nosso servlet container. Similar a @GET temos também @POST, @DELETE, @PUT entre outras anotações para cada método HTTP...

Perceba que estamos falando de URI, mas a URL final dependerá do contexto da sua aplicação no servidor:
{servidor}/{contexto da aplicação}/{URI do seu recurso}

Como último passo vamos configurar o RESTEasy para reconhecer automaticamento o recurso e instalar ele para receber as requisições HTTP. Isso é feito no famigerado arquivo web.xml(encontrado em ?WebContent/WEB-INF"), melhor explicado nessa página. Abaixo temos o nosso novo web.xml e você pode copiar o código abaixo para substituir o conteúdo do seu atual.

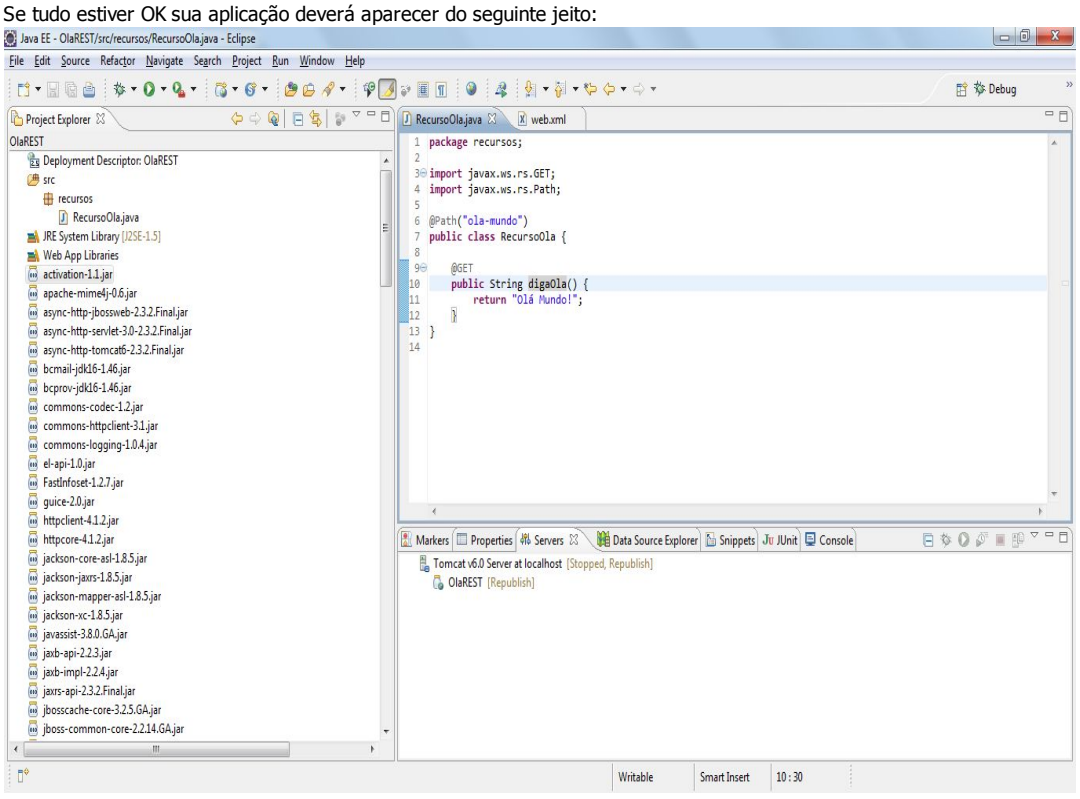
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   id="WebApp_ID" version="2.5">
   <display-name>OlaMundo</display-name>

   <!-- RESTEasy: mapeie "TODOS" meus recursos JAX-RS -->
   <context-param>
      <param-name>resteasy.scan</param-name>
      <param-value>true</param-value>
   </context-param>

   <!-- O Servlet do RESTeasy pega as requests para direcionar para as suas
      classes JAX-RS -->
   <servlet>
      <servlet-name>Resteasy</servlet-name>
      <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
   </servlet>

   <!-- O servlet do RESTEasy responde na raiz da aplicação -->
   <servlet-mapping>
      <servlet-name>Resteasy</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Se tudo estiver OK sua aplicação deverá aparecer do seguinte jeito:



Agora é hora então de testar!

- Clique em cima do seu servidor tomcat que você criou anteriormente na aba Server;
- Clique em iniciar ou segure CTRL+ALT e aperte R

Note os logs do tomcat no console e entre eles você deverá perceber a seguinte linha:

INFO: Server startup in 503 ms

Quando isso aparecer, abra o navegador e acesse a URL http://localhost:8080/OlaREST/ola-mundo. O resultado deverá ser o texto ?Olá Mundo!" no seu navegador

\o/
Bem, na verdade isso foi chato e entediante. Tanto trabalho para um ?Olá Mundo!" no navegador?! Agora vamos fazer algo mais interessante e as coisas vão ficar mais rápidas!

Pegando parâmetros da requisição HTTP

JAX-RS possibilita através de anotações pegar parâmetros da sua URI e do cabeçalho da requisição HTTP. Os parâmetros pode ser parâmetros que usamos na seguinte forma:

{url}?parametro=valor

Ou também parâmetros pegos na própria URI, exemplo:

pessoa/{id}

Onde {id} seria um valor a ser passado para nosso método.

As anotações para esse fim são respectivamente @QueryParam e @PathParam>. Mexemos um pouco no código do nosso RecursoOla para aceitar ambos parâmetros:

```
package recursos;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;

@Path("ola-mundo")
public class RecursoOla {

   @GET
   public String digaOla(@QueryParam("parametro") String parametro) {
      return "Olá Mundo! <br /> O parâmetro de \"Query\"enviado foi: " + parametro;
   }

   @GET
   @Path("/{parametro}")
   public String digaOlaPath(@PathParam("parametro") String parametro) {
      return "Olá Mundo! <br /> O parâmetro de \"URI\" enviado foi: " + parametro;
   }
}
```

OPA! @Path em cima de método, e agora? Se você usar @Path em cima de método, isso quer dizer que iremos ?aninhar" as URIs. Ou seja, se você tentar acessar com GET a URI /ola-mundo, o método **digaOla** será chamado. Se tentar acessar a URI /ola-mundo/qualquer%20texto%20aqui, o método **digaOlaPath** será chamado. No caso do primeiro método você ainda pode informar um parâmetro na forma /ola-mundo?parametro=algum%20valor
Modifique seu código e faça o processo de ?subir" o tomcat de novo, mas agora acesse novas URLs no seu navegador. Abaixo a URL a ser acessada e o resultado esperado:

http://localhost:8080/OlaREST/ola-mundo/ESTE%20%C3%89%20UM%20TESTE

Olá Mundo!
O parâmetro de "URI" enviado foi: ESTE É UM TESTE

http://localhost:8080/OlaREST/ola-mundo/?parametro=ESTE%20%C3%89%20OUTRO%20TESTE!

Olá Mundo!
O parâmetro de "Query"enviado foi: ESTE É OUTRO TESTE!

Agora você aprendeu o básico do básico do JAX-RS. Mas sabemos que aplicações da ?vida real" não são só troca de texto e ?Olá Mundos", né? No mundo real usamos formatos como XML, JSON, CSV. ?Usamos Beans que vem do banco de dados, mas espostos por um EJB que usa JPA para acessar um Oracle e tudo isso em um ambiente de alta avaliabilidade usando cluster de servidores de aplicações com replicação de estado?. Bem, vamos por favor continuar no Olá Mundo, mas agora iremos nos divertir um pouco mais :-)

Retornando suas classes nos recursos JAX-RS

Nessa parte iremos criar um REST WS mais avançado. No final do artigo temos dois downloads um que contém um projeto do Eclipse com o que foi feito até agora e outro com todo o projeto.
Digamos que você tem um bean para representar o seu ?Olá Mundo!" e seu sistema quer mostrar essa mensagem em diversos idiomas. E mais: Você quer permitir o usuário escolha o formato que vai ser exibido a mensagem! Agora a coisa pegou! Na verdade seria pior se JAX-RS e RESTEasy não tivessem sido inventados, mas com eles a coisa fica fácil e fácil!

Primeiramente vamos modelar as URIs de acessos ao nosso serviço:

URI	Método HTTP	O que faz?	Formato retornado	Parâmetro
/ola-mundo	GET	Retorna todas as mensagens	XML, JSON	tamanho e cor(só para o formato HTML)
/ola-mundo/{idioma}	GET	retorna a mensagem de acordo com o idioma	XML, JSON, HTML	tamanho, cor(só para o formato HTML) e idioma(parâmetro de URI)

Basicamente vamos servir os recursos em três formatos: XML, JSON e HTML. O formato HTML aceita uns parâmetros de formatação. Obviamente vai além do escopo desse artigo discutir se isso é legal fazer em um WS REST ou não, simplesmente iremos demonstrar a habilidade de lidar com diversos formatos e fica a cargo do leitor entender corretamente os príncipios REST.
Em seguida, o que vamos fazer é criar um pacote para nossa classe, eu criei ?modelo" e dentro desse pacote criei minha classe OlaMundo, como você pode ver abaixo:

```
package modelo;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class OlaMundo {
   private String idioma;
   private String mensagem;

   public OlaMundo() {

   }
   public OlaMundo(String idioma, String mensagem) {
      super();
      this.idioma = idioma;
      this.mensagem = mensagem;
   }
   public String getIdioma() {
      return idioma;
   }
   public void setIdioma(String idioma) {
      this.idioma = idioma;
   }
   public String getMensagem() {
      return mensagem;
   }
   public void setMensagem(String mensagem) {
      this.mensagem = mensagem;
   }
}
```

Perceba que estamos usando nossa classe semelhante as classes chamadas de ?JavaBeans", pois contemos os atributos privados e métodos de acesso à eles. A importância de criar classes usando o padrão Java Beans é que essas classes poderão ser usadas com diversos frameworks, como JPA, JAXB
A anotação @XmlRootElement sobre a classe é importante para que o RESTEasy escolha um MessageBodyWriter para essa classe, em outras palavras, com o ela o RESTeasy vai saber criar o corpo HTTP da resposta. Como respeitamos o padrão JavaBeans, essa classe poderá sem problemas ser ?parseada" pelo JAXB (usando pelo RESTEasy ?por baixo dos panos").

Para lidar com essa OlaMundo vamos criar uma classe dentro do pacote servicos chamada ServicoOlaMundo. Ela é responsável por lidar com as nossas mensagens gravadas.

Você deve estar se perguntando por que criar uma classe se podíamos lidar com OlaMundo diretamente na classe JAX-RS. Estamos criar um Web Service REST aqui, em uma aplicação comumente não temos só os serviços WEB, mas também, por exemplo, páginas JSP, JSF, um cliente JavaFX que fazem parte da aplicação. Se você joga toda a lógica no REST ws, ela não será reaproveitada em outras partes!.

O código de nossa classe se encontra abaixo:

```
package servicos;
```

```java
import java.util.ArrayList;
import java.util.List;

import modelo.OlaMundo;

/**
 * @author William Antônio
 *
 */
public class ServicoOlaMundo {

   public List<OlaMundo> buscaTodos() {
      return OlaMundoBD.buscaTodos();
   }

   public OlaMundo buscaPorIdioma(String idioma) {
      return OlaMundoBD.buscaPorIdioma(idioma);
   }

   /**
    * Essa class estática simula um banco de dados de OlaMundo's
    *
    * @author William Antônio
    *
    */
   public static class OlaMundoBD {
      private static List<OlaMundo> mensagens;

      static {
         mensagens = new ArrayList<OlaMundo>();
         mensagens.add(new OlaMundo("PT", "Olá Mundo!"));
         mensagens.add(new OlaMundo("EN", "Hello World!"));
         mensagens.add(new OlaMundo("ES", "¡Hola Mundo!"));
         mensagens.add(new OlaMundo("DE", "Hallo Welt!"));
      }

      /**
       * Todas as mensagens
       *
       * @return
       */
      public static List<OlaMundo> buscaTodos() {
         return mensagens;
      }

      /**
       * Tenta buscar por idioma, retorna nulo se não encontrada a mensagem no
       * idioma que pedimos
       *
       * @param idioma
       * @return
       */
      public static OlaMundo buscaPorIdioma(String idioma) {
         for (OlaMundo olaMundo : mensagens) {
            if (olaMundo.getIdioma().equals(idioma))
               return olaMundo;
         }
         return null;
      }

   }
}
```

Os mais experientes podem se queixar de como feia ficou essa classe! Nosso foco aqui não é o serviço REST, então criamos uma simples classe para simplesmente simular o acesso a esse recurso. Não iremos falar dela, mas sim usá-la :-)

Agora é a hora mais interessante! Vamos mexer no RecursoOlaMundo para acessar esse serviço e fazer o que propomos um pouco mais acima na nossa tabelinha. Abaixo a minha classe já terminada e no próximo parágrafo uma explicação dos pontos novos e mais interessantes marcados por comentários no código.

```java
package recursos;

import java.util.List;

import javax.ws.rs.DefaultValue;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response.Status;

import org.jboss.resteasy.annotations.providers.jaxb.Wrapped;

import modelo.OlaMundo;
import servicos.ServicoOlaMundo;

@Path("ola-mundo")
public class RecursoOla {

   private ServicoOlaMundo servicoOlaMundo;
```

```
    public RecursoOla() {
       //1
       servicoOlaMundo = new ServicoOlaMundo();
    }

    @GET
    //2
    @Wrapped(element = "mensagensOlaMundo")
    //3
    @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
    public List<OlaMundo> buscaTodos() {
       return servicoOlaMundo.buscaTodos();
    }

    //4
    @Path("/{idioma: [A-Z][A-Z]}")
    @GET
    @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
    public OlaMundo buscaPorIdioma(@PathParam("idioma") String idioma) {
       OlaMundo res = servicoOlaMundo.buscaPorIdioma(idioma);
       //5
       if (res == null)
          throw new WebApplicationException(Status.NOT_FOUND);
       return res;
    }

    @Path("/{idioma: [A-Z][A-Z]}")
    @GET
    //6
    @Produces({ MediaType.TEXT_HTML })
    public String buscaPorIdiomaHTML(@PathParam("idioma") String idioma,
          @QueryParam("cor") @DefaultValue("red") String cor,
          @QueryParam("tamanho") @DefaultValue("12") int tamanho) {
       OlaMundo res = servicoOlaMundo.buscaPorIdioma(idioma);
       if (res == null)
          throw new WebApplicationException(Status.NOT_FOUND);
       StringBuffer sb = new StringBuffer();
       sb.append("<span style='");
       sb.append("color:" + cor + ";");
       sb.append("font-size:" + tamanho + "px;");
       sb.append("'>");
       sb.append(res.getMensagem());
       sb.append("</strong>");
       return sb.toString();
    }
}
```

A primeira coisa a ser notada em (1) é a instanciação do ServicoOlaMundo dentro do construtor do nosso Recurso. Antes que você se pergunte, saiba que a cada request uma instância dessa classe será criada e jogada fora logo após ela servir a requisição. Quem conhece os príncipios REST sabe do famoso ?Stateless": A comunicação entre o cliente e o servidor deve ser sem estado. Basicamente uma instância de ServicoOlaMundo será criada com cada requisição HTTP que chega e também descartada ao final. Isso não importa, por que nosso banco de dados fictício é ?static" e não tem instância, assim os dados serão os mesmo sempre e mantidos em memória ;)
O comentário marcando número 2 mostra a anotação proprietária do RESTeasy chamada @Wrapped. Ela server para informar o nome que você quer que sua collection apareça na listagem.

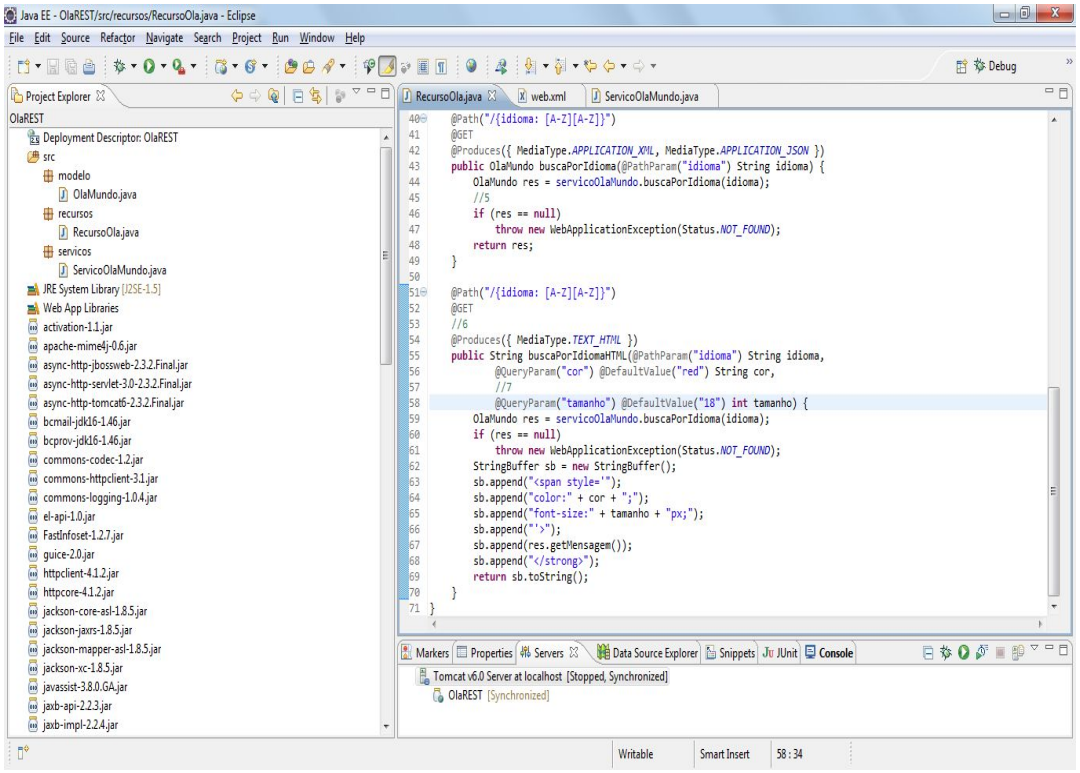
O comentário número 3 destaca talvez a característica mais espetacular de toda a especificação. O mesmo método serve dois formatos completamente distintos e fica a cargo da implementação(no caso o RESTEasy) arruma uma forma de gerar o objeto retornado no formato certo. Mas como RESTeasy fará isso? O cabeçalho HTTP ?Accepts", presente em todas as requisições HTTP, irá informar isso, ou seja, se você quer XML, você terá XML, se quiser JSON, terá JSON. Se pedir um formato não suportado um código de erro apropriado será lançado na resposta do servidor.

No 4 temos um simples, mas útil recurso fornecido pelo JAX-RS. Você pode utilizar REGEX para determinar a URI. Se a requisição for para um URI que não se encaixa nesse REGEX, o servidor lançará um 404(não encontrado) para você.
O comentário 5 destaca algo que fizemos para manter a simplicidade. Se uma mensagem para o idioma não for encontrada, lançamos 404. Podíamos também lançar uma ?exceção de negócio" e mapear ela para o 404, mas isso fica para outro dia ;D

O 6 aumenta a ?espetacularidade" do descrito no comentário 3. Observe que os métodos buscaPorIdioma e buscaPorIdiomaHTML respondem pela mesma URI. Como RESTeasy irá escolher o método apropriado então? Pela cabeçalho Accepts da requisição feita pelo cliente!! Ou seja, se o cliente ?pedir" HTML, ele vai cair nesse método, se ele pedir JSON ou XML, ele cairá no outro.

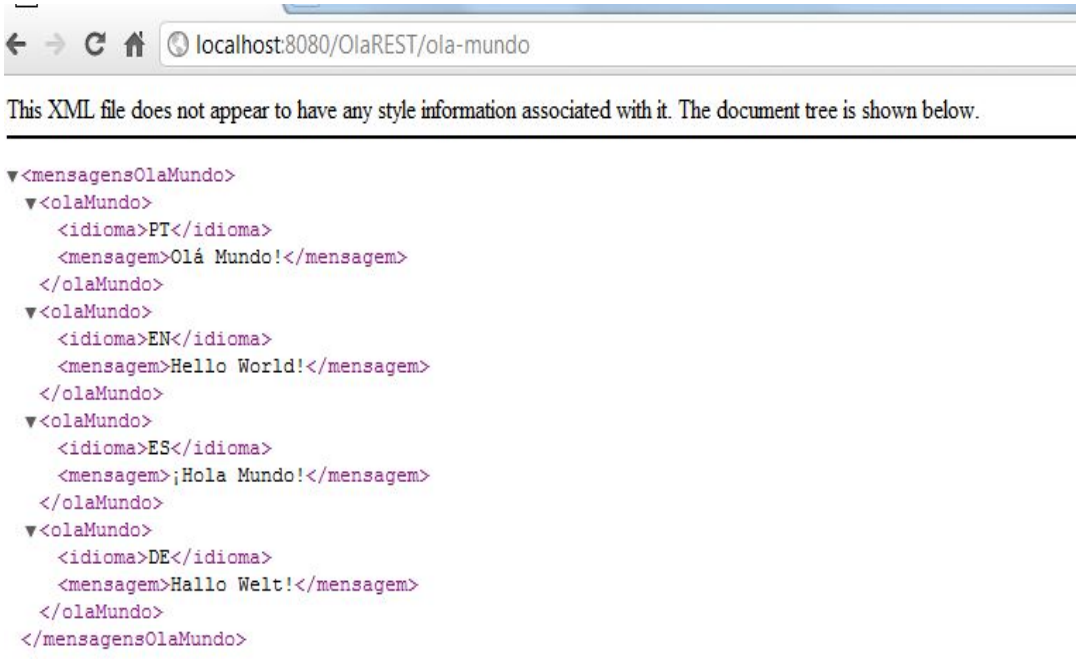
Por fim, o 7 mostra uma outra anotação muito interessante do JAX-RS. O @DefaultValue permite que você escolha um valor padrão se um parâmetro não for fornecido pelo cliente. Isso se encaixa bem no nosso caso quando o cliente for pedir HTML.
O corpo do método buscaPorIdiomaHTML deveria ser diferente, pois no caso do método buscaPorIdioma RESTeasy irá fazer o parse do objeto para JSON ou XML (tudo isso por causa da anotação @XmlRootElement na classe OlaMundo), no caso de HTML RESTeasy não sabe o que fazer e temos duas abordagens: A nossa(que é realizar o parse no corpo do método JAX-RS), mas ela é feia e imatura(mas serve para fins didáticos) ou então poderíamos criar o nosso próprio MessageBodyWriter para ensinar para o RESTeasy como parsear o objeto para HTML. Então ele escolheria nossa classe de parse e usaria quando a request fosse para HTML.

Ufa! Vamos testar? Se tudo correu certo, a estrutura do seu projeto deve estar semelhante a essa:

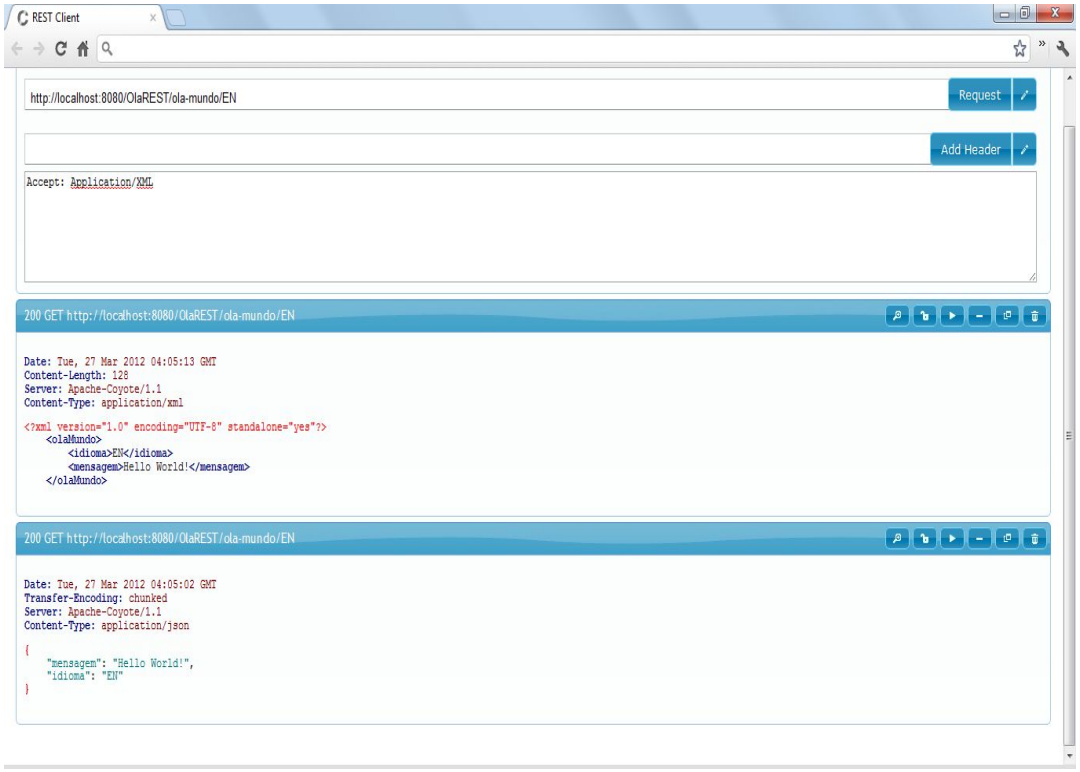


Restart o servidor Tomcat (ou republique sua aplicação) e vá para o navegador para realizar um novo teste. Se acessar a URL http://localhost:8080/OlaREST/ola-mundo/ES?cor=RED&tamanho=300 um letreiro gigante em vermelho escrito ?¡Hola Mundo!" deverá aparecer.

Se acessar http://localhost:8080/OlaREST/ola-mundo/, um XML deverá aparecer listando todas as mensagens:



Mas e os formatos XML e JSON quando quisermos selecionar uma mensagem usando o idioma? Bem, você deve ter uma ferramenta que faça requisições HTTP e permita você mexer no cabeçalho para mudar o tipo de formato que você quer (JSON, XML, HTML ou application/json, application/xml e text/html). Abaixo usei uma extensão(REST Client) para o Google e consegui pegar os dados no formato desejado.



Conclusão

Apresentamos os conceitos básicos de JAX-RS para criação de WEB Services REST(também chamados RESTful) usando Java. É claro que há muito mais para aprender. Se o você quiser saber mais sobre JAX-RS recomendamos o Livro RESTful with JAX-RS ou a leitura da própria especificação. Se quiser saber mais sobre RESTEasy você pode ler esse artigo que saiu na Java Magazine 99.