argonavis
tecnologia e arte

serviços web RESTful JAX-RS

Helder da Rocha

JAVA EE 7

O texto foi elaborado como material de apoio para treinamentos especializados em linguagem Java e explora assuntos detalhados nas especificações e documentações oficiais sobre o tema, utilizadas como principais fontes. A autoria deste texto é de inteira responsabilidade do seu autor, que o escreveu independentemente com finalidade educativa e não tem qualquer relação com a Oracle.

O código-fonte relacionado aos tópicos abordados neste material estão em:

`github.com/helderdarocha/javaee7-course`
`github.com/helderdarocha/CursoJavaEE_Exercicios`
`github.com/helderdarocha/ExercicioMinicursoJMS`
`github.com/helderdarocha/JavaEE7SecurityExamples`

www.argonavis.com.br

R672p Rocha, Helder Lima Santos da, 1968-

Programação de aplicações Java EE usando Glassfish e WildFly.

360p. 21cm x 29.7cm. PDF.

Documento criado em 16 de outubro de 2013.

Atualizado e ampliado entre setembro e dezembro de 2016.

Volumes (independentes): *1: Introdução, 2: Servlets, 3: CDI, 4: JPA, 5: EJB, 6: SOAP, 7: REST, 8: JSF, 9: JMS, 10: Segurança, 11: Exercícios*.

1. Java (*Linguagem de programação de computadores*). 2. Java EE (*Linguagem de programação de computadores*). 3. Computação distribuída (*Ciência da Computação*). I. Título.

CDD 005.13'3

Capítulo 7: Web Services REST

1 Introdução

1.1 Arquitetura REST

REST significa **Re**presentational **S**tate **T**ransfer. É um **estilo arquitetônico** baseado na World Wide Web. Consiste de recursos (páginas) em arquitetura que facilita a transferência de estado no cliente (navegação via links, hierarquias, métodos HTTP) usando representação uniforme (URI, tipos MIME, representações). A World Wide Web é uma implementação de arquitetura REST. As especificações HTTP 1.1 e URI (Uniform Resource Identifier) foram escritas em acordo com a arquitetura REST.

Web Services baseados em REST são chamados de RESTful Web Services e adotam a arquitetura REST, representação de URIs, tipos MIME, HTTP e restrições (estado, cache, etc) para fornecer intraestrutura que permite um ambiente de computação distribuída eficiente, simples e com overhead mínimo.

1.2 O protocol HTTP (RFC 2616)

O protocol HTTP é um protocolo de transferência de dados TCP/IP, síncrono, baseado na estratégia requisição-resposta e sem estado.

Uma mensagem de requisição consiste de uma linha inicial composta de método HTTP, URI e versão, zero ou mais linhas de cabeçalho propriedade:valor, seguidos de uma linha em branco e um corpo opcional (dependente do método usado). Por exemplo:

xx

Os métodos **HTTP/1.1** são GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT.

Uma mensagem de resposta consiste de uma linha inicial composta de versão, código de status e mensagem, linhas de cabeçalho propriedade:valor, seguidos por uma linha em branco e um corpo opcional (dependente do método usado). Por exemplo:

xx

Os **códigos de status** são: 1xx (informação), 2xx (sucesso), 3xx (redirecionamento), 4xx (erro no cliente) e 5xx (erro no servidor).

1.3 Arquitetura de WebServices usando REST

RESTful Web Services aproveitam a infraestrutura de um website para utilizer como interface CRUD para dados disponibilizados como recursos HTTP. A arquitetura de RESTful WebServices consiste de:

- recursos (páginas), árvore de navegação (links, hierarquia)
- representações de dados e tipos (URIs, MIME)
- vocabulário de operações do protocolo HTTP (GET, POST, ...)

Os dados anexados a respostas de serviços REST podem utilizar diversas representações e tipos diferentes. Os formatos mais comuns são XML, JSON, CSV, etc.

As URIs representam objetos, com estado e relacionamentos aninhados. Por exemplo, um objeto dentro de uma aplicação Java contendo relacionamentos representados por

pais.estado.cidade

poderia ser representado em REST pela URI:

http://servidor/aplicacao/**pais/estado/cidade/**

Usando métodos HTTP, é possível realizar operações CRUD usando URIs. Esta é a essência da arquitetura de WebServices REST:

- Create: **POST /pais/estado** (contendo <**estado**>**SP**</**estado**>, no corpo)
- Retrieve: **GET /pais/estado/SP** (Retrieve All com **GET /pais/estado**)
- Update: **PUT /pais/estado/PB**
- Delete: **DELETE /pais/estado/PB**

2 Serviços RESTful em Java com JAX-RS

2.1 Arquitetura do JAX-RS

JAX-RS é uma API Java para facilitar o desenvolvimento de aplicações que utilizam a arquitetura REST. Seus principais componentes são:

- Uma subclasse de **Application**, usada para configurar a aplicação, mapear o nome do contexto e listar outras classes que fazem parte da aplicação (opcional).
- **Root resource classes**, classes mapeadas a XML ou JSON que definem a raiz de um recurso mapeado a um caminho de URI
- **Resource methods**, métodos de um Root resource class, que são mapeados automaticamente a **métodos** HTTP GET, POST, etc.

- **Providers**, operações que produzem ou consomem representações de entidades em outros formatos (ex: XML, JSON)

JAX-RS usa anotações para configurar esses componentes

- **@ApplicationPath** na subclasse de Application
- **@Path** nos Root resource classes
- **@GET, @POST,** etc e **@Path** nos Resource methods
- **@Produces, @Consumes** nos Providers
- **@Context** para injetar diversos tipos de contexto disponíveis no ambiente Web

2.2 Uma aplicação HelloWorld com JAX-RS

Aplicações REST podem ser construídos com aplicações Web (WARs) ou com Session Beans (EJB). Para construir um serviço RESTful simples com JAX-RS empacotado em um WAR, siga os passos abaixo.

1. Crie uma subclasse de **Application** vazia anotada com **@ApplicationPath** informando um nome para o contexto Web do serviço (este nome aparecerá na URL, dentro do contexto da aplicação Web):

```
@ApplicationPath("webapi")
public class HelloApplication extends Application { }
```

2. Crie um root resource anotado com **@Path** (este nome também aparecerá na URL, dentro do contexto do ApplicationPath):

```
@Path("sayhello")
public class HelloResource {
  …
}
```

3. Crie um método no resource anotado com designador de um método HTTP:

```
@GET
public String getResposta() {
    return "Hello, world!";
}
```

4. Empacote a aplicação em uma WAR (ex: **helloapp.war**) e faça o deploy em um servidor de aplicações ou servidor Web que suporte JAX-RS (ex: **localhost:8080**)
5. Envie GET para a URL abaixo (pode ser via browser) e veja a resposta

```
http://localhost:8080/helloapp/webapi/sayhello
```

Pela interface do browser é possível testar apenas método **GET** e envio de **text/plain**. Testar outros métodos e representações de dados é difícil. Nesses casos pode-se criar um cliente HTTP (usando Apache Commons HTTP, por exemplo), ferramentas de linha de comando que permitem gerar comandos HTTP (ex: cURL) ou ferramentas disponíveis nos browsers. Alguns exemplos são Mozilla **Firebug**, Chrome Dev Tools, ou ferramentas especializadas como o Firefox RESTClient.

3 Configuração do serviço

A configuração da URI base pode ser no web.xml:

```
<servlet-mapping>
   <servlet-name>javax.ws.rs.core.Application</servlet-name>
   <url-pattern>/webapi/*</url-pattern>
</servlet-mapping>
```

ou via anotação **@ApplicationPath** em subclasse de Application

```
@ApplicationPath("/webapi")
public class ApplicationConfig extends Application { ... }
```

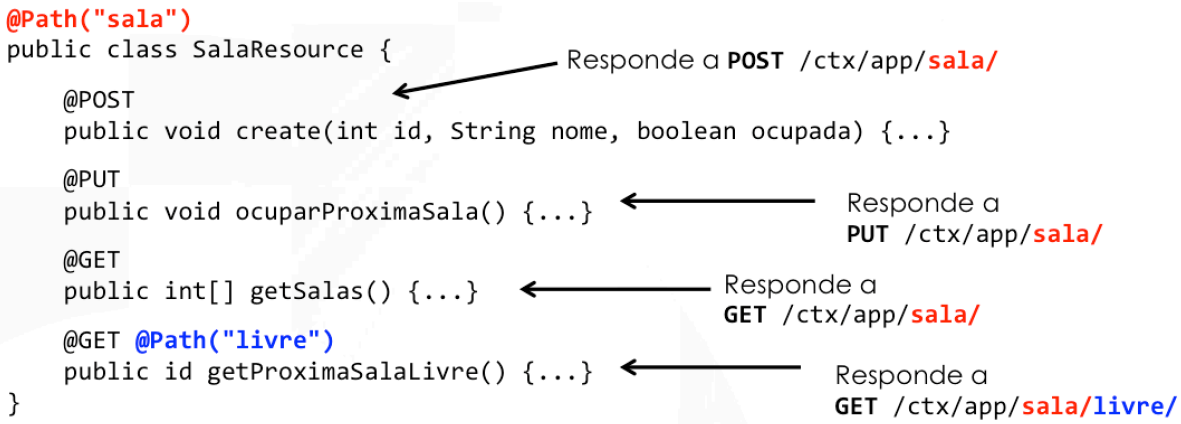
Application carrega todos os resources encontrados. Pode-se sobrepor getClasses() para selecionar quais disponibilizar

```
@javax.ws.rs.ApplicationPath("webapi")
public class ApplicationConfig extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        resources.add(com.argonavis.festival.FilmeResource.class);
        resources.add(com.argonavis.festival.SalaResource.class);
        return resources;
    }
```

4 Construção de resources

4.1 Anotação @Path e @PathParam

@Path mapeia um **caminho** à resource raiz ou método. Pode ser especificado antes de uma **classe** ou **método**. Se um método não expliticamente definir um @Path próprio, ele responde ao @Path declarado na classe onde está contido. Se método define um @Path e também há um @Path na classe, ele é um **subcontexto** do @Path raiz. Não deve haver ambiguidade nos mapeamentos resultantes.



Um **@Path** pode receber parâmetros entre chaves. Os parâmetros são chamados de path templates:

```
@Path("/filmes/{imdb}")
```

Isto permite que a URI correspondente aceite qualquer coisa depois de filmes/, por exemplo: http://abc.com/war/app/filmes/tt0066921. O parâmetro pode ser associado a uma variável local acessível no método usando **@PathParam**. No exemplo abaixo, o valor passado na URL pode ser lido pela variável local codigoIMDB:

```
@Path("/filmes/{imdb}")
public class FilmesIMDBResource {
   @GET  @Produces("text/xml")
   public Filme getFilme(@PathParam("imdb") String codigoIMDB) {
      return entity.getFilmeByIMDBCode(codigoIMDB);
   }
}
```

Se a URI passada não combinar com a resolução do path template, o servidor produzirá um **erro 404.** O string passado ao template é bastante restrito e deve combinar com a expressão regular **"[^/]+?"** mas é possível **substituir** a expressão regular default por outra:

```
@Path("filme/{imdb: tt[0-9]{4,7}}")
```

Um template também pode ter mais de uma variável. Cada uma deve ser mapeada a uma variável local usando @PathParam:

```
@Path("{cdd1:[0-9]}/{cdd2:[0-9]}")
public class AssuntoResource {
    @GET @Path("{cdd3:[0-9]}")
    pubic void getAssunto(@PathParam("cdd1") int d1,
                          @PathParam("cdd2") int d2,
                          @PathParam("cdd3") int d3) { ... }
```

```
...
}
```

O padrão acima combina com GET /ctx/app/**5/1/0**, por exemplo.

Assim como em qualquer URI, espaços e caracteres especiais devem ser substituídos por URL encodings:

`/cidade/São Paulo → /cidade/S%E3o%20Paulo`

4.2 Designadores de métodos HTTP

Designadores de métodos são anotações com os nomes dos métodos HTTP, e servem para mapear métodos HTTP a métodos do resource. Os designadores disponíveis são @GET, @POST, @PUT, @DELETE e @HEAD mas podem ser criados outros designadores se necessário.

Os métodos mapeados devem retornar dados compatíveis com a operação do método mapeado. Se retornarem **void** e não receberem mensagens com corpo, não precisam de nenhuma configuração adicional, mas se precisam retornar um corpo para a mensagem (ex: GET), ou se precisam extrair dados do corpo da mensagem e não apenas da URI (ex: POST) é necessário configurar um entity provider.

Um entity provider é uma implementação de MessageBodyReader ou MessageBodyWriter e é configurado usando uma anotação @Produces ou @Consumes. Para tipos comuns (strings XML ou JSON) existem providers prontos. Se for necessário configurar cabeçalhos de resposta para um determinado método, pode-se usar um objeto javax.ws.rs.core.**Response**:

4.3 Idempotência em REST

No contexto de uma aplicação REST, um método é **idempotente** se, quando chamado múltiplas vezes produz os mesmos resultados. Os métodos idempotentes do HTTP são GET, HEAD, PUT e DELETE. POST não é idempotente já que chamadas sucessivas do mesmo método poderão produzir resultados diferentes.

Ao mapear métodos HTTP a métodos do resource, deve-se observar se o método do resource também é idempotente, para manter a consistência de comportamento, ou seja, deve-se usar **POST** para criar novos dados (C), usar **GET** apenas para retornar dados (R), usar **PUT** apenas para alterar dados (U) e usar **DELETE** apenas para remover dados (D).

4.4 @GET

@GET pode ser usado para retornar representações do resource. A representação **default** retornada é sempre **text/plain**. Outras representações MIME podem ser configuradas usando um entity provider através da anotação **@Produces**:

```
@GET
@Produces("text/html")
public String getHtml() {
    return "<html><body><h1>Hello!!</h1></body></html>";
}
```

4.5 @DELETE

@DELETE deve mapeado a métodos responsáveis pela remoção de instâncias do resource:

```
@DELETE
@Path({id})
public void deleteAssento(@PathParam("id") int id) {
    facade.removeAssento(id);
}
```

4.6 @PUT e @POST

POST e PUT podem ser usados para create e update. A decisão é questão de estilo e discussões acadêmicas. Como PUT é idempotente, é geralmente usado para fazer *atualizações* do resource, mas updates usando PUT **não podem ser parciais** (é preciso enviar o objeto inteiro).

Nos exemplos deste curso, usaremos **@POST** para *criar* resources:

```
@POST @Consumes({"application/xml", "application/json"})
public void create(Sala entity) {
    facade.create(entity);
}
```

E **@PUT** para realizar *updates*:

```
@PUT @Path("{id}")
@Consumes({"application/xml", "application/json"})
public void edit(@PathParam("id") Long id, Sala entity) {
      facade.update(entity);
}
```

5 Provedores de entidades

Os provedores de entidades fornecem **mapeamento** entre diferentes **representações** e tipos Java. JAX-RS disponibiliza provedores para diversos tipos Java e várias representações populares (texto, XML, JSON, etc.)

Provedores são **selecionados** nos métodos com anotações **@Produces** e **@Consumes**, indicando um tipo **MIME** que está **mapeado ao provedor.** É possível criar provedores para representações não suportadas através da implementação de **MessageBodyWriter** (para produção) e **MessageBodyReader** (para consumo).

O provedor implementa a interface, é anotado com **@Provider** e mapeia o tipo que representa através de **@Produces** ou **@Consumes**

```
@Provider @Produces("application/msword")
public class WordDocWriter implements MessageBodyWriter<StreamingOutput> { … }
@GET @Path("arquivo.doc") @Produces({"application/msword"})
public StreamingOutput getWordDocument() { … }
```

5.1 MediaTypes

Tipos MIME suportados em HTTP podem ser representados por strings ou por constantes de **MediaType**. Não faz diferença. vantagem de usar uma constante é a verificação em tempo de compilação. Algumas constantes de MediaType são: APPLICATION_JSON, APPLICATION_FORM_URLENCODED, APPLICATION_XML, TEXT_HTML, TEXT_PLAIN, MULTIPART_FORM_DATA.

As anotações abaixo são equivalentes

```
@Consumes({"text/plain,text/html"})
@Consumes({MediaType.TEXT_PLAIN,MediaType.TEXT_HTML})
```

5.2 Provedores nativos

Não é preciso criar **@Provider** para representações de vários tipos Java que são mapeados automaticamente. Os tipos byte[], String, InputStream, Reader, File e javax.activation.DataSource suportam todos os tipos MIME (*/*). Os tipos javax.xml.transform.Source e javax.xml.bind.JAXBElement (classes geradas via JAXB) suportam application/xml, application/json e text/xml. O tipo MultivaluedMap<String, String> suporta dados de formulário (application/x-www-form-urlencoded).

Para outros tipos há StreamingOutput, que suporta streaming de todos os tipos MIME na resposta (através de um MessageBodyWriter). Por exemplo, para streaming de imagens, videos, PDF, RTF, etc.

5.3 @Produces e @Consumes

@Produces é usado para mapear tipos MIME de representações de resource **retornado** ao cliente:

```
@GET @Produces({"application/xml"})
public List<Filme> findAll() {…}
```

Um **cliente** HTTP pode selecionar o método com base nos tipos MIME que ele suporta através de um cabeçalho HTTP de requisição "**Accept:**"

```
GET /ctx/app/filmes
Accept: application/xml, application/json
```

@Consumes é usado para mapear tipos MIME de representações de resource **enviado** ao serviço:

```
@POST @Consumes({"application/xml", "application/json"})
public void create(Filme entity) { … }
```

Um **cliente** HTTP pode enviar um tipo MIME compatível com o método desejado usando um cabeçalho de requisição "**Content-type:**"

```
POST /ctx/app/filmes
Content-type: application/xml
```

As anotações podem ser declaradas no nível da classe, estabelecendo um valor **default** para todos os métodos. Métodos individuais podem sobrepor valores herdados:

```
@Path("/myResource")
@Produces("application/xml")
@Consumes("text/plain")
public class SomeResource {
   @GET
   public String doGetAsXML() { ... }
   @GET @Produces("text/html")
   public String doGetAsHtml() { … }
   @PUT @Path({text})
   public String putPlainText(@PathParam("text") String txt) { … }
}
```

6 Builders

6.1 ResponseBuilder

ResponseBuilder permite construir e configurar a resposta HTTP enviada ao cliente. Um **Response** cria um código de status que retorna **ResponseBuilder**, que possui diversos

métodos para construir cabeçalhos, códigos de resposta, criar cookies, anexar dados, etc. Seus métodos também devolvem ResponseBuilder permitindo uso em cascata. Ao final, o método terminal **build()** retorna o Response finalizado.

```
ResponseBuilder builder = Response.status(401);
Response r1 = builder.build();
Response r2 = Response.ok().entity(filme).build();
Response r3 = Response.status(200)
                      .type("text/html")
                      .cookie(new NewCookie("u","1"))
                      .build();
```

6.2 URIBuilders

URIs são a interface mais importante em aplicações REST, mas podem ser strings complexos, principalmente se tiverem muitos parâmetros. Uma UriBuilder pode ser usada para construir uma java.net.URI corretamente a partir de seus componentes individuais, segmentos e fragmentos, evitando erros devido a URIs mal-formadas.

```
// cria a URI resultado#ancora;abc=xyz?allow=sim
URI requestUri = UriBuilder.fromPath("{arg1}")
                           .fragment("{arg2}")
                           .matrixParam("abc", "xyz")
                           .queryParam("allow","{arg3}")
                           .build("resultado", "ancora", "sim");
```

7 Requisição

Além de **@PathParam** há outras cinco anotações que permitem extrair informação de um request:

- **@QueryParam** e **@DefaultValue** Extraem dados de um query string (?nome=valor&nome=valor)
- **@FormParam** Extrai dados de formulário (application/x-www-form-urlencoded)
- **@CookieParam** Extrai dados de cookies (pares nome=valor)
- **@HeaderParam** Extrai dados de cabeçalhos HTTP
- **@MatrixParam** Extrai dados de segmentos de URL

7.1 QueryParam e DefaultValue

Parâmetros de query são anotados com duas anotações:

@QueryParam("parametro") extrai dados passados no query-string de uma URI, O argumento deve informar o nome do parametro HTTP recebido que será associado a uma variável local do método.

@DefaultValue("valor") deve informar o valor que será usado caso o parâmetro não tenha sido definido.

Por exemplo, a requisição:

```
GET /ctx/app/filme?maxAno=2010&minAno=1980
```

Recebida pelo método abaixo:

```
@GET @Path("filme")
public List<Filme> getFilmesPorAno(
    @DefaultValue("1900") @QueryParam("minAno") int min
    @DefaultValue("2013") @QueryParam("maxAno") int max)   {...}
```

irá associar 1980 à variável min e 2010 à variável max, mas esta outra requisição também pode ser usada:

```
GET /ctx/app/filme
```

Neste caso, como os parâmetros não foram definidos na URI, serão usados os valores default min=1900 e max=2013.

7.2 FormParam

@FormParam extrai parâmetros de formulários HTML. Por exemplo, o formulário abaixo recebe três parâmetros:

```
<FORM action="http://localhost:8080/festival/webapi/filme"
      method="post">
   Titulo: <INPUT type="text" name="titulo" tabindex="1">
   Diretor: <INPUT type="text" name="diretor" tabindex="2">
   Ano: <INPUT type="text" name="ano" tabindex="3">
</FORM>
```

Um deles é consumido pelo método abaixo:

```
@POST
@Consumes("application/x-www"form-urlencoded")
public void post(@FormParam("diretor") String diretor) {
   ...
}
```

7.3 HeaderParam

@HeaderParam permite obter dados que estão no cabeçalho da requisição (ex: Content-type, User-Agent, etc.). Por exemplo, a requisição HTTP abaixo:

```
GET /ctx/app/filme/echo HTTP/1.1
Cookies: version=2.5, userid=9F3402
```

Recebida pelo resource:

```
@Path("/filme")
public class Filme {

  @GET
  @Path("/echo")
  public Response echo(@HeaderParam("Cookie") String cookieList) {
       return Response.status(200)
                  .entity("You sent me cookies: " + cookieList)
                  .build();
   }
}
```

Produzirá a resposta abaixo:

```
You sent me cookies: userid=9F3402, version=2.5
```

7.4 CookieParam

@CookieParam permite acesso a cookies individuais que o cliente está mandando para o servidor. Por exemplo, a requisição HTTP:

```
GET /ctx/app/filme HTTP/1.1
Cookies: version=2.5, userid=9F3402
```

Recebida pelo resource abaixo:

```
@GET
@Produces({MediaType.TEXT_PLAIN})
public Response getMyCookies(
            @CookieParam(value = "userid") String userid,
            @CookieParam(value = "version") String version) {
     return Response.status(200)
                  .entity("Your cookies contain: " + userid " and " + version).build();
}
```

Irá produzir a resposta:

```
Your cookies contain: 9F3402 and 2.5
```

7.5 MatrixParam

@MatrixParam extrai os pares nome=valor que são incluídos como anexo da URL (geralmente usado como alternativa a cookies). Por exemplo, a requisição HTTP

```
GET /ctx/app/filme;version=2.5;userid=9F3402
```

Recebida pelo resource:

```
@GET
@Produces({MediaType.TEXT_PLAIN})
public Response getMyCookies(
            @MatrixParam(value = "userid")  String userid,
            @MatrixParam(value = "version") String version) {
     return Response.status(200)
                    .entity("Your URL contains: " + userid + " and " + version).build();
}
```

Irá produzir a resposta:

```
Your URL contains: 9F3402 and 2.5
```

8 Validação

Os dados enviados para métodos em classes de resource podem ser validados através da **API Bean Validation**, que é configurada via anotações.

```
@POST
@Path("/criar")
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public void criarFilme(
     @NotNull   @FormParam("titulo")  String titulo,
     @NotNull   @FormParam("diretor") String diretor,
     @Min(1900) @FormParam("ano")     int ano) { … }
@Pattern(regexp="tt[0-9]{5-7}")
private String imdbCode;
```

Violações na validação provocam **ValidationException** que gera um erro **500** (Internal Server Error) se a exeção ocorreu ao validar um tipo de retorno ou gera um erro **400** (Bad Request) se a exeção ocorreu ao validar um parâmetro de entrada.

9 Objetos de contexto

9.1 @Context

@Context pode ser usado para injetar diversos objetos contextuais disponíveis em uma requisição ou resposta HTTP. Isto inclui os seguintes objetos da Servlet API: ServletConfig, ServletContext, HttpServletRequest e HttpServletResponse, além dos objetos Application, UriInfo, Request, HttpHeaders, SecurityContext e Providers da JAX-RS API.

```
@Context
Request request;
@Context
UriInfo uriInfo;
@PUT
public metodo(@Context HttpHeaders headers) {
   String m = request.getMethod();
   URI ap   = uriInfo.getAbsolutePath();
   Map<String, Cookie> c = headers.getCookies();
}
@GET @Path("auth")
public login(@Context SecurityContext sc) {
    String userid =
           sc.getUserPrincipal().getName();
    (if sc.isUserInRole("admin")) { … }
}
```

9.2 UriInfo e HttpHeaders

Pode-se ler vários parâmetros de uma vez usando os objetos UriInfo e HttpHeaders. A interface **UriInfo** contém informações sobre os componentes de uma requisição, e a interface **HttpHeaders** contém informações sobre cabeçalhos e cookies. Ambas podem ser injetadas atraves de **@Context:**

```
@GET
public String getParams(@Context UriInfo ui,
                        @Context HttpHeaders hh) {
    MultivaluedMap<String, String> queryParams =
                           ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams =
                           ui.getPathParameters();
    MultivaluedMap<String, String> headerParams =
                           hh.getRequestHeaders();
    MultivaluedMap<String, Cookie> pathParams =
                           hh.getCookies();
}
```

10 Segurança

10.1 Autenticação HTTP

A **autenticação** HTTP é realizada por cabeçalhos. Por exemplo, um cliente que tentasse acessar dados protegidos em um servidor poderá obter a seguinte resposta do servidor:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="localhost:8080"
```

A resposta, ao ser recebida por um cliente, forçaria a abertura de uma janela para entrada de credenciais. Depois que o usuário digitar as credenciais, ela talvez seja enviado pelo cliente através de uma requisição como a seguinte:

```
GET /ctx/app/filmes HTTP/1.1
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Se a autenticação falhar, o servidor responde com **403 Forbidden**

A configuração da autenticação deve ser realizada no servidor. Arquivo web.xml permite definir método de autenticação: **BASIC**, **DIGEST**, **CLIENT-CERT** (HTTPS). A escolha do método de autenticação é feito usando o tag <**login-config**>

```
<login-config>
  <auth-method>
    BASIC
  </auth-method>
</login-config>
```

10.2 Autorização

Ferramentas do servidor definem perfis (**roles**) ou grupos mapeados a usuários autenticados. **Roles** são mapeados a diferentes URIs no **web.xml** e precisam ser declarados:

```
<security-role>
   <role-name>admin</role-name>
</security-role>
<security-role>
   <role-name>membro</role-name>
</security-role>
<security-role>
   <role-name>visitante</role-name>
</security-role>
```

A declaração de perfis de usuários é feita no web.xml usando <**security-role**>. A associação desses perfis com usuários reais é dependente de servidor (ex: grupos no JBoss grupos = roles).

10.3 Configuração de acesso

A coleção de recursos Web protegidos e métodos de acesso que podem ser usados para acessá-los é definido em um bloco <**web-resource-collection**> definido dentro de <**security-constraint**>**.** O bloco inclui URL-patterns <**url-pattern**> que indicam quais os mapeamentos que abrangem a coleção e um <**http-method**> para cada método permitido:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Seção de Assinantes</web-resource-name>
    <url-pattern>/aplicacao/filme/*</url-pattern>
    <url-pattern>/aplicacao/sala/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  ...
</security-constraint>
```

O Security Constraint também associa perfis (roles) de usuário (declarados previamente) à coleção através de itens <auth-constraint>. Há também um item para especificar proteção aos dados usando SSL (no exemplo abaixo garantindo confidencialidade):

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Administracao</web-resource-name>
    <url-pattern>/aplicacao/filme/*</url-pattern>
    ...
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
...
<security-constraint> outras coleções, … </security-constraint>
```

Veja o *Capítulo 10 Segurança* para mais detalhes sobre configuração de segurança.

11 Cliente RESTful

Um cliente JAX-RS pode ser **qualquer cliente HTTP** (java.net.*, Apache HTTP Client, cURL, etc.) Pode ser escrito em Java, JavaScript, C#, Objective-C ou qualquer linguagem capaz de abrir **sockets** de rede. É preciso lidar com as **representações** recebidas: converter

XML, JSON, encodings, etc., além de gerar e interpretar os **cabeçalhos HTTP** usados na comunicação para seleção de tipos MIME, autenticação, etc.

Uma alternativa é utilizar frameworks que possuem APIs para escrever clientes:

- Jersey: http://jersey.java.net
- RESTEasy: http://www.jboss.org/resteasy

Existe também uma **API padrão**, disponível a partir do JAX-RS 2.0 (Java EE 7).

11.1 Usando comunicação HTTP nativa

Qualquer API capaz de montar requisições HTTP pode ser usada para implementar clientes. O exemplo abaixo utiliza HttpURLConnection (do Java SE 7) para enviar um comando GET, configurar o tipo de dados a ser aceito como resposta (XML) e lidar com erros:

```
URL url =
   new URL("http://localhost:8080/ctx/app/imdb/tt0066921");
HttpURLConnection conn =
   (HttpURLConnection) url.openConnection();
conn.setRequestMethod("GET");
conn.setRequestProperty("Accept", "application/xml");
if (conn.getResponseCode() != 200)  {
   throw new RuntimeException("Erro : " + conn.getResponseCode());
}

BufferedReader br =
    new BufferedReader(new InputStreamReader((conn.getInputStream())));
String linha = br.readLine();
System.out.println("Dados recebidos: " + linha);
conn.disconnect();
JAXBContext jc = JAXBContext.newInstance(Filme.class);
Unmarshaller u = jc.createUnmarshaller();
Filme filme = (Filme) u.unmarshal(new StringReader(linha));

System.out.println(filme.getIMDB()); …
```

11.2 Cliente Apache HttpClient

Usar a API do Apache HttpComponents HTTP Client é ainda mais simples. O exemplo abaixo faz o mesmo usando esta API:

```
GetMethod method =
   new GetMethod("http://localhost:8080/ctx/app/filme/imdb/tt0066921");
method.addRequestHeader("Accept", "application/xml");
```

```
HttpClient client = new HttpClient();
int responseCode = client.executeMethod(method);
if (responseCode != 200)  {
   throw new RuntimeException("Erro : " + responseCode);
}
String response = method.getResponseBodyAsString();
JAXBContext jc = JAXBContext.newInstance(Filme.class);
Unmarshaller u = jc.createUnmarshaller();
Filme filme = (Filme) u.unmarshal(new StringReader(linha));

System.out.println(filme.getIMDB()); …
```

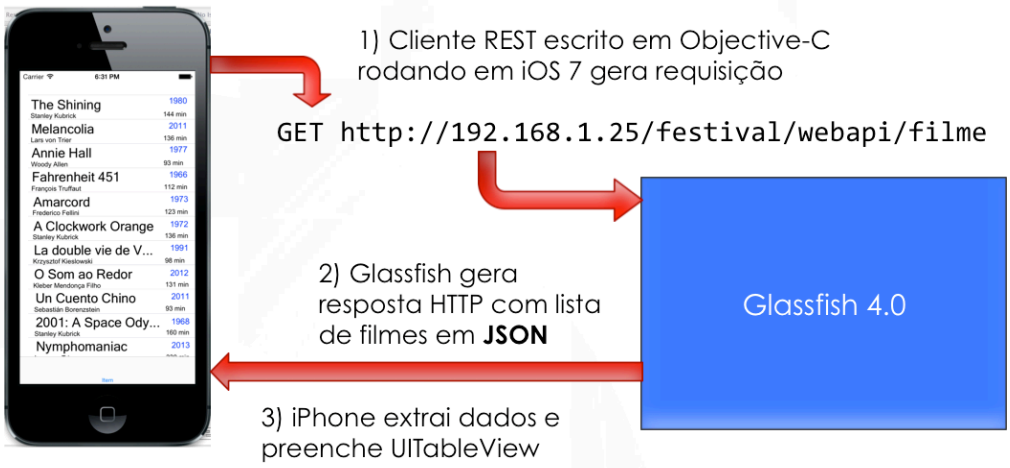
11.3 Usando APIs RESTEasy e Jersey

Uma API de cliente específica para Web Services REST como o Jersey facilita o trabalho convertendo automaticamente as representações em objetos (não é preciso recorrer ao JAXB):

```
ClientConfig config =
    new DefaultClientConfig();
Client client = Client.create(config);
URI baseURI =
    UriBuilder.fromUri("http://localhost:8080/ctx").build();
WebResource service = client.resource(baseURI);
Filme filme = service.path("app")
              .path("filme/imdb/tt0066921")
              .accept(MediaType.APPLICATION_XML)
              .get(Filme.class);
System.out.println(filme.getIMDB());
System.out.println(filme.getTitulo());
System.out.println(filme.getDiretor());
```

11.4 Clientes RESTful em aplicativos Web e mobile

REST é a melhor alternativa para Web Services que fazem **integração** como outras plataformas. Por exemplo, um cliente Objective-C ou Swift rodando em iOS acessando um serviço JAX-RS:



11.5 Cliente JAX-RS 2.0

Java EE 7 possui uma API padrão para clientes REST baseada na API do Jersey. O trecho de código abaixo demonstra o envio de uma requisição GET simples:

```
Client client = ClientBuilder.newClient();
String nomeDoFestival =
     client.target("http://localhost:8080/festival/webapi/nome")
           .request(MediaType.TEXT_PLAIN)
           .get(String.class);
```

Pode-se configurar o cliente com **WebTarget** para reusar o path base:

```
Client client = ClientBuilder.newClient();
WebTarget baseResource = client.target("http://servidor/ctx/app");
WebTarget filmeResource = base.path("filme");
```

A partir do resource pode-se construir um **request()** e enviar um método HTTP:

```
Filme filme = filmeResource.queryParam("imdb", "tt0066921")
                           .request(MediaType.APPLICATION_XML)
                           .get(Filme.class)
```

12 WADL

Para descrever serviços REST e permitir a geração automática de clientes, há o padrão **WADL** - Web Application Description Language que tem função análoga ao WSDL (usada em SOAP Web Services) para descrição de Web Services RESTful.

Um WADL de uma aplicação instalada pode ser gerado automaticamente e obtido em http://servidor/contexto/raiz/**application.wadl**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
  <grammars/>
```

```
<resources base="http://localhost:8080/festival/webapi">
    <resource path="filme/{imdb}">
       <param type="xs:string" style="template" name="imdbID"/>
       <method name="GET" id="getFilme">
            <response>
                <representation mediaType="application/xml"/>
                <representation mediaType="application/json"/>
            </response>
        </method>
…
```

JAX-RS 2.0 (Java EE 7) fornece uma ferramenta de linha de comando (e task Ant/Maven) que gera código para escrever um cliente JAX-RS usando WADL. A instrução

```
wadl2java -o diretorio
          -p pacote
          -s jaxrs20
          http://localhost:8080/ctx/raiz/application.wadl
```

cria uma classe (JAXBElement) para cada resource raiz e uma classe Localhost_CtxRaiz.class (para um serviço em http://localhost/ctx/raiz). Esta classe possui um método createClient() que cria um cliente JAX-RS 2.0 já configurado para usar o serviço. Ela também fornece métodos para retornar instâncias e representações dos objetos mapeados como resource.

13 Referências

13.1 Especificações

[1] HTTP: http://www.w3.org/Protocols/

[2] RFC 2616 (HTTP) http://www.ietf.org/rfc/rfc2616.txt

[3] WADL https://wadl.java.net/

[4] Arquiteturas de Web Services http://www.w3.org/TR/ws-arch/

13.2 Artigos, documentação e tutoriais

[5] Saldhana. "*Understanding Web Security using web.xml via Use Cases*" Dzone. 2009. http://java.dzone.com/articles/understanding-web-security

[6] *Java EE Tutorial sobre RESTful WebServices* http://docs.oracle.com/javaee/7/tutorial/doc/jaxrs.htm

[7] *Documentação do Jersey (tutorial de JAX-RS)*: https://jersey.java.net/documentation/latest

[8] Roy Thomas Fielding. "*Architectural Styles and the Design of Network-based Software Architectures*". University of California, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[9] Alex Rodriguez. "*RESTful Web Services: the basics*". IBM Developerworks, 2008. http://www.ibm.com/developerworks/webservices/library/ws-restful/

[10] Hadyel & Sandoz (editors). "*JAX-RS: Java API for RESTful Web Services. Version 1.1*. Oracle, 2009. https://jax-rs-spec.java.net/