serviços web RESTful com JAX-RS

Helder da Rocha (helder@acm.org)

Atualizado em dezembro de 2014

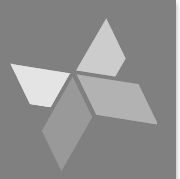
argonavis
tecnologia e arte

JAVA EE 7

Sobre este tutorial

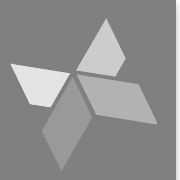
Este é um tutorial sobre tecnologia JAX-RS (de acordo com a especificação Java EE 7) criado para cursos presenciais e práticos de programação por Helder da Rocha

O código dos exemplos está disponível em repositório público no GitHub e é software livre sob os termos da licença Apache 2.0.



Conteúdo

1. Introdução a REST e JAX-RS
2. Resources em JAX-RS
3. Clientes JAX-RS w WADL
4. Segurança



JAVAEE7

JAX-RS

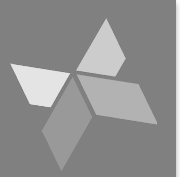
RESTful web services

1

introdução a rest e jax-rs

REST (Fielding, W3C TAG, 2000)

- **Re**presentational **S**tate **T**ransfer
  - **Estilo arquitetônico** baseado na World Wide Web
  - **Recursos** (páginas) em arquitetura que facilita a transferência de estado no cliente (links, hierarquias, métodos HTTP) usando representação uniforme (URI, tipos MIME, representações)
  - A World Wide Web é uma implementação de arquitetura REST
  - Especificações HTTP 1.1 e URI foram escritas em acordo com REST
- **Web Services** baseados em REST (**RESTful WS**)
  - Adotam a arquitetura REST, representação de URIs, tipos MIME, HTTP e restrições (estado, cache, etc) que fornecem intraestrutura para um ambiente de computação distribuída eficiente, simples e com overhead mínimo



HTTP/1.1 (RFC 2616)

- Protocolo de requisição-resposta sem estado
- Mensagem de **requisição** consiste de `GET /index.html HTTP/1.1`
  - Linha inicial composta de **método**, **URI** e **versão**
  - Linhas de cabeçalho **propriedade: valores** + linha em branco
  - **Corpo** opcional (depende do método)
  - Métodos: **GET**, **HEAD**, **POST**, **PUT**, **DELETE**, **TRACE**, **OPTIONS**, **CONNECT**
- Mensagem de **resposta** consiste de `HTTP/1.1 200 OK`
  - Linha inicial composta de **versão**, **código de status** e **mensagem**
  - Linhas de cabeçalho **propriedade: valores** + linha em branco
  - **Corpo** opcional (depende do método)
  - Códigos de status da resposta: **1xx** (informação), **2xx** (sucesso), **3xx** (redirecionamento), **4xx** (erro no cliente) e **5xx** (erro no servidor)

RESTful Web Services

- Aproveitam a infraestrutura de um website
  - Recursos (páginas), árvore de navegação (links, hierarquia)
  - Representações de dados e tipos (URIs, MIME)
  - Vocabulário de operações do protocolo HTTP (GET, POST, …)
- URIs representam **objetos** (com relacionamentos aninhados)
  - Um objeto contendo relacionamentos

    `pais.estado.cidade`

    de uma **aplicação** pode ser representado em REST pela URI

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

RESTful Web Services

- Dados anexados a respostas podem ter diferentes representações e tipos
  - XML, JSON, CSV, etc.
- Métodos HTTP permitem operações CRUD
  - Create: **POST** `/pais/estado` (`<estado>SP</estado>`, no corpo)
  - Retrieve: **GET** `/pais/estado/SP` (Retrieve all: **GET** `/pais/estado`)
  - Update: **PUT** `/pais/estado/PB`
  - Delete: **DELETE** `/pais/estado/PB`

Como implementar em Java

- Pode-se criar resources usando **WebServlets**
  - Métodos doGet(), doPost(), doPut(), doDelete() para lidar com diferentes métodos HTTP
  - Métodos de HttpServlet getServletPath() e getPathInfo() para lidar com diferentes contextos da URI
  - HttpServletRequest para ler cabeçalhos da requisição e extrair dados, HttpServletResponse para montar respostas
  - Contextos e listeners para configurar provedores, conversores, etc.
- **API JAX-RS** permite criar resources com POJOs anotados

JAX-RS

- API Java para desenvolver serviços RESTful
- Componentes de uma aplicação JAX-RS
  - **Subclasse de Application**: configuração da aplicação: mapeia nome do contexto e lista classes que fazem parte da aplicação
  - **Root resource class**: define a raiz de um recurso mapeado a um caminho de URI
  - **Resource method**: métodos de um Root resource class associados a designadores de métodos HTTP (@GET, @POST, etc.)
  - **Providers**: operações que produzem ou consomem representações de entidades em outros formatos (ex: XML, JSON)

JAX-RS

- JAX-RS usa **anotações** para configurar os componentes
  - **@ApplicationPath** em subclasse de Application
  - **@Path** em Root resource classes
  - **@GET**, **@POST**, etc e **@Path** em Resource methods
  - **@Produces**, **@Consumes** em Providers
  - **@Context** para injetar diversos tipos de contexto disponíveis no ambiente Web

Exemplo: HelloWorld com JAX-RS

- 1. Crie uma subclasse de **Application** vazia e anote com **@ApplicationPath** informando o nome do contexto

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

- 2. Crie um **root resource** anotado com **@Path**

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

Exemplo: HelloWorld com JAX-RS

- 3. Crie um método no resource, anotado com um **designador de método HTTP** (compatível com o método)

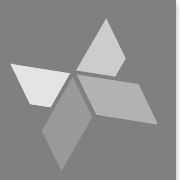
```
@GET
public String getResposta() {
    return “Hello, world!”;
}
```

- 4. Empacotar em um **WAR** (ex: **helloapp.war**) e **implantar** (ex: localhost:8080) em um servidor Web (que suporte JAX-RS)
- 5. Envie **GET** para a URL abaixo (pode ser via browser) e veja a resposta

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

Como testar RESTful Web Services

- Pela interface do browser é possível testar **GET** e envio de **text/plain**
- Para testar outros métodos use um **cliente HTTP**
  - Linha de comando: **cURL**
  - Mozilla **Firebug**, Firefox **RESTClient**, Chrome **Dev Tools**, etc.
  - **Java REST Client** https://code.google.com/p/rest-client/
  - **Fiddler** (no Windows) http://fiddler2.com/
  - **MitmProxy** (no Linux, OS X) http://mitmproxy.org/
- Escreva testes unitários usando uma API HTTP
  - Apache HTTP Components – **HTTP Client** http://hc.apache.org/

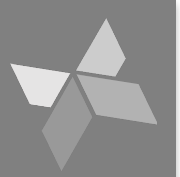


Exercícios

- 1. Configure seu ambiente e **crie uma aplicação REST simples** (como o exemplo HelloWorld mostrado) usando JAX-RS. Experimente usando acesso simples em um browser e um cliente REST (ex: FireBug).

- 2. Crie métodos adicionais com **@GET** configurando anotações **@Path()** diferentes em cada um. Ex: **@GET @Path("/um")** e **@GET @Path("/dois")** para acessar **/hello/um** e **/hello/dois** retornando strings diferentes.

- 3. Escreva um resource REST similar ao do exercício anterior **sem usar JAX-RS**. Use um WebServlet e um método **doGet()** que identifique a URI usando **getPathInfo()**.



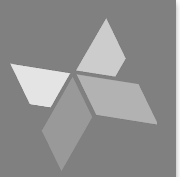
JAVAEE7

JAX-RS

RESTful web services

2

resources jax-rs



Configuração do serviço

- Configuração da **URI base** pode ser no **web.xml**

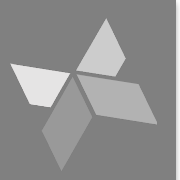
```
<servlet-mapping>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
    <url-pattern>/app/*</url-pattern>
</servlet-mapping>
```

- Ou via anotação **@ApplicationPath** em subclasse de **Application**

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

- Application carrega **todos** os resources (default) ou uma **seleção**

```
@javax.ws.rs.ApplicationPath("app")
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;
    }
}
```



Anotação @Path

- Mapeia um caminho a resource raiz ou método
  - Se método não define **@Path** próprio, ele responde ao **@Path** declarado na classe
  - Se método define um **@Path**, ele é **subcontexto** do **@Path** raiz

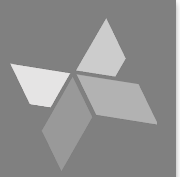
```
@Path("sala")
public class SalaResource {

    @POST
    public void create(int id, String nome, boolean ocupada) {...}

    @PUT
    public void ocuparProximaSala() {...}

    @GET
    public int[] getSalas() {...}

    @GET @Path("livre")
    public id getProximaSalaLivre() {...}
}
```

Responde a POST **/ctx/app/sala/**

Responde a PUT **/ctx/app/sala/**

Responde a GET **/ctx/app/sala/**

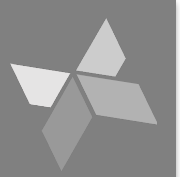
Responde a GET **/ctx/app/sala/livre/**



Path templates e @PathParam

- **@Path** pode receber parâmetros (**Path templates**)
  - @Path("/filmes/**{imdb}**")
  - Ex: http://abc.com/war/app/filmes/**tt0066921**
- **@PathParam** associa o parâmetro a um argumento de método

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



Path template e restrições

- Se URL for incompatível com o path template, o servidor retorna erro **404**
- Variável do template combina (default) com a expressão regular "**[^/]+?**" mas pode ser substituída

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

- Um template pode ter mais de uma variável

```
@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) { ... }
}
```

Combina com
GET /ctx/app/5/1/0

- Espaços e caracteres especiais devem ser substituídos por URL encodings

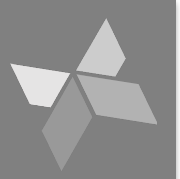
```
/cidade/São Paulo -> /cidade/S%E3o%20Paulo
```

Designadores de métodos HTTP

- Mapeia métodos HTTP a métodos do resource
  - **@GET, @POST, @PUT, @DELETE, @HEAD**
  - Podem ser criados outros designadores
- Métodos mapeados podem retornar
  - **void**: não requer configuração adicional
  - **Tipos Java (objeto ou primitivo)**: tipo deve ser convertido por entity provider (**MessageBodyReader** ou **MessageBodyWriter**) configurável via **@Produces** ou **@Consumes**
  - **Um objeto javax.ws.rs.core.Response**: configura metadados da resposta (ex: cabeçalhos)

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** (chamadas sucessivas podem produzir resultados diferentes)
- Deve-se manter a **consistência** de comportamento observando, ao fazer o mapeamento, se o método do resource também é idempotente
  - Usar **POST** para criar novos dados (**C**)
  - Usar **GET** apenas para retornar dados (**R**)
  - Usar **PUT** apenas para alterar dados (**U**)
  - Usar **DELETE** apenas para remover dados (**D**)



@GET

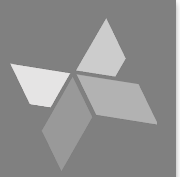
- **@GET** pode ser usado para retornar representações do resource
  - Representação default retornada é text/plain
  - Outras representações MIME podem ser configuradas usando um entity provider através da anotação @Produces

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

@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);
}
```



@PUT e @POST

- **POST** e **PUT** podem ser usados para **Create** e **Update**
  - PUT é idempotente, portanto é recomendado para updates do resource (updates não devem ser parciais: é preciso enviar o objeto inteiro)
- Neste 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);
}
```

Provedores de entidades

- Fornecem **mapeamento** entre diferentes **representações** e **tipos** Java
  - Suportam representações populares (texto, XML, JSON, etc.)
  - São **selecionados** nos métodos com anotações **@Produces** e **@Consumes**, indicando um tipo MIME mapeado ao provedor

```
@GET @Path("arquivo.doc") @Produces({"application/msword"})
public StreamingOutput getWordDocument() { … }
```

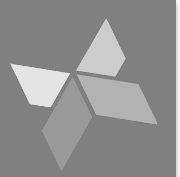
- Pode-se **criar** provedores para outras representações implementando **MessageBodyWriter** (produção) e **MessageBodyReader** (consumo)
  - O provedor implementa a interface anotada com **@Provider** e mapeando o tipo que representa via **@Produces** ou **@Consumes**

```
@Provider @Produces("application/msword")
class WordDocWriter implements MessageBodyWriter<StreamingOutput> { … }
```

MediaType

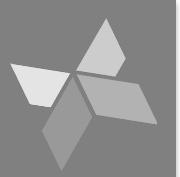
- Tipos MIME suportados em HTTP podem ser representados por **strings** ou por **constantes** de **MediaType** (verificadas em tempo de compilação)
- Algumas constantes
  - **APPLICATION_JSON**
  - **APPLICATION_FORM_URLENCODED**
  - **APPLICATION_XML**
  - **TEXT_HTML**
  - **TEXT_PLAIN**
  - **MULTIPART_FORM_DATA**
- As anotações abaixo são equivalentes

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



Provedores nativos

- Não é preciso **@Provider** para tipos Java mapeados automaticamente
- Suportam todos os tipos MIME (***/***): **byte[]**, **String**, **InputStream**, **Reader**, **File** e **javax.activation.DataSource**
- Suportam **application/xml**, **application/json** e **text/xml:**
  - **javax.xml.transform.Source** e **javax.xml.bind.JAXBElement**
- Suporta dados de formulário (**application/x-www-form-urlencoded**)
  - **MultivaluedMap<String, String>**
- Streaming de todos os tipos MIME na resposta (**MessageBodyWriter**)
  - **StreamingOutput** (imagens, videos, PDF, RTF, etc.)



@Produces

- **@Produces** é usado para mapear tipos MIME de representações de resource retornado ao cliente
  - `@GET @Produces({"application/xml"})`
    `public List<Filme> findAll() {…}`
- **Cliente HTTP** pode selecionar método com base nos tipos MIME que ele suporta através de um cabeçalho de requisição **Accept:**
  - `GET /ctx/app/filmes`
    `Accept: application/xml, application/json`

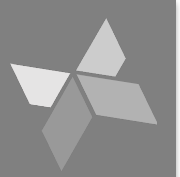
@Consumes

- **@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) { … }`
- **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`

@Produces e @Consumes

- Podem ser declaradas no nível da classe, estabelecendo um valor default para todos os métodos
- Métodos individuais podem sobrepor valores herdados

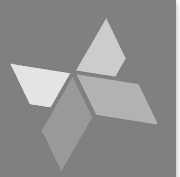
```
@Path("/myResource")
@Produces("application/xml")
@Consumes("text/plain")
public class SomeResource {
    @GET
    public String doGetAsXML() { ... }

    @GET @Produces("text/html")
    public String doGetAsHtml() { … }

    @PUT @Path({text})
    public String putPlainText(@PathParam("text") String txt) { … }
}
```



Response e ResponseBuilder

- Constrói a resposta HTTP enviada ao cliente
- **Response** cria um código de status que retorna **ResponseBuilder**
- **ResponseBuilder** possui métodos para construir cabeçalhos, códigos de resposta, criar cookies, anexar dados, etc.
- Métodos devolvem **ResponseBuilder** permitindo uso em cascata
- Método **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("user","argo"))
                      .build();
```



java.net.URI e URIBuilders

- URIs são a interface mais importante em aplicações REST
- Um **java.net.URI** pode ser construído usando um **UriBuilder**
  - Permite trabalhar com os componentes individuais, segmentos e fragmentos de uma URI, concatenar e construir novas URIs

```
// cria a URI /index.html#top;p1=abc;p2=xyz?allow=yes?id=123
URI requestUri =
    UriBuilder.fromPath("{arg1}")
              .fragment("{arg2}")
              .matrixParam("p1", "abc")
              .matrixParam("p2", "{arg3}")
              .queryParam("allow","{arg4}")
              .queryParam("id", "123")
              .build("/index.html", "top", "xyz", "yes");
```

Parâmetros do request

- 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**

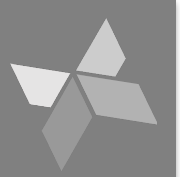
QueryParam e DefaultValue

- **@QueryParam("parametro")** – extrai dados passados no query-string de uma URI. O argumento informa o nome do parametro HTTP que será associado ao argumento do método
- **@DefaultValue("valor")** – informa um valor que será usado caso o parâmetro não tenha sido definido
- Exemplo de requisição

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

- Exemplo de método que lê os parâmetros

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



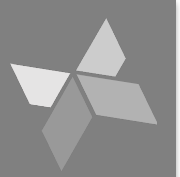
FormParam

- **@FormParam** extrai parâmetros de formulários HTML
- Exemplo de formulário

```
<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>
```

- Exemplo de método que consome dados do formulário

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

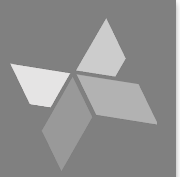


HeaderParam

- **@HeaderParam** permite obter dados que estão no cabeçalho da requisição (ex: Content-type, User-Agent, etc.)
- Requisição HTTP
  - `GET /ctx/app/filme/echo HTTP/1.1`
    `Cookies: version=2.5, userid=9F3402`
- Resource

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

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

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

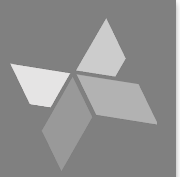


CookieParam

- **@CookieParam** permite acesso a cookies individuais que o cliente está mandando para o servidor
- Requisição HTTP
  - GET /ctx/app/filme HTTP/1.1
    Cookies: **version**=2.5, **userid**=9F3402
- Resource

```
@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();
}
```

Your cookies contain:
9F3402 and 2.5

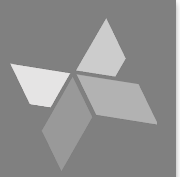


MatrixParam

- **@MatrixParam** extrai os pares **nome=valor** que são incluídos como anexo da URL (usado como alternativa a cookies)
- Requisição HTTP
  - GET /ctx/app/filme;**version**=2.5;**userid**=9F3402
- 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 URLs contains: "
                           + userid + " and " + version).build();
}
```

Your URL contains:
9F3402 and 2.5



UriInfo e HttpHeaders

- Permite ler vários parâmetros de uma vez
  - **UriInfo** contém informações sobre componentes de uma requisição
  - **HttpHeaders** contém informações sobre cabeçalhos e cookies
  - As duas interfaces podem ser injetadas atraves de **@Context**

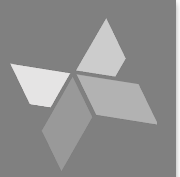
```
@GET
public String getParams(@Context UriInfo ui,
                        @Context HttpHeaders hh) {
    MultivaluedMap<String, String> queryParams =
                                    ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams =
                                    ui.getPathParameters();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

Validação com Bean Validation

- Os dados enviados para métodos em classes de resource podem ser validados com a **API Bean Validation**, 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**
  - Erro **500** (Internal Server Error) se ocorreu ao validar tipo de retorno
  - Erro **400** (Bad Request) se ocorreu ao validar parâmetro de entrada



@Context

- **@Context** pode ser usado para **injetar** objetos contextuais disponíveis em uma requisição ou resposta HTTP
- Objetos da **Servlet API**:
  - ServletConfig,
  - ServletContext,
  - HttpServletRequest
  - HttpServletResponse
- Objetos da **JAX-RS API**:
  - Application
  - UriInfo
  - Request
  - HttpHeaders
  - SecurityContext
  - Providers

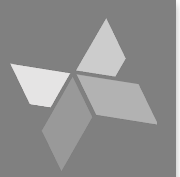
```
@Context Request request;
@Context UriInfo uriInfo;

@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")) { … }
}
```

Exercícios

- 4. Escreva um REST Web Service para acesso a uma lista de **Produtos**. Planeje a interface e ofereça operações para
  - **Listar** todos os produtos
  - **Detalhar** um produto
  - **Criar** um produto
  - **Alterar** o preço de um produto
  - **Pesquisar** produtos por faixa de preço
  - **Remover** um produto
- 5. Crie métodos que retornem JSON ou XML
- 6. Use um cliente REST para testar a aplicação e selecionar o tipo (via cabeçalho Accept)

Produto

`id: long`
`descricao: string`
`preco: double`



JAVAEE7

JAX-RS

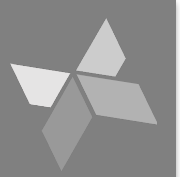
RESTful web services

3

clientes jax-rs

Cliente REST

- 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.
  - É preciso gerar e interpretar **cabeçalhos HTTP** usados na comunicação (seleção de tipos MIME, autenticação, etc.)
- Alternativas em **Java**
  - **Jersey**: http://jersey.java.net
  - **RESTEasy**: http://www.jboss.org/resteasy
  - API padrão, a partir do **JAX-RS 2.0** (Java EE 7)



Cliente usando java.net

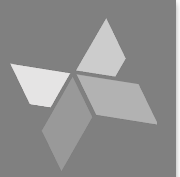
- Qualquer API capaz de montar requisições HTTP pode ser usada para implementar clientes

```
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()); …
```

Cliente Apache HttpClient

- API do Apache HttpComponents **HTTP Client** é ainda mais simples

```
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()); …
```



Cliente Jersey*

- Uma API de cliente específica para Web Services REST como o **Jersey** facilita o trabalho convertendo automaticamente representações em objetos

```
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());
```

* Outra alternativa é a API **RESTEasy**

Cliente mobile (iPhone)

- REST é a melhor alternativa para Web Services que fazem integração como outras plataformas
- Exemplo: cliente em iPhone usando Objective-C

1) Cliente REST escrito em Objective-C rodando em iOS 7 gera requisição

`GET http://192.168.1.25/festival/webapi/filme`

Glassfish 4.0

2) Glassfish gera resposta HTTP com lista de filmes em **JSON**

3) iPhone extrai dados e preenche UITableView

Cliente JAX-RS 2.0

- **Java EE 7** tem uma **API padrão** para clientes REST baseada no **Jersey**
- Exemplo: 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 com **WebTarget** para **reusar** o **path base**

```
Client client = ClientBuilder.newClient();
WebTarget base = client.target("http://localhost:8080/festival/webapi");
WebTarget filmeResource = base.path("filme");
```

- Do resource pode-se criar um **request()** e enviar um **método HTTP**

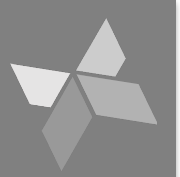
```
Filme filme = filmeResource.queryParam("imdb", "tt0066921")
                           .request(MediaType.APPLICATION_XML)
                           .get(Filme.class)
```

JAVA EE 7

WADL: Web App Description Language

- Descreve serviços REST (e permitir a geração automática de clientes)
- Análoga ao WSDL (usado em serviços SOAP)
- Obtida em http://**servidor**/**webctx**/**restctx**/**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>
...
```

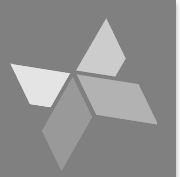


Geração de clientes

- **JAX-RS 2.0** (Java EE 7) fornece uma **ferramenta de linha de comando** (e task Ant/Maven) que gera um cliente JAX-RS usando WADL

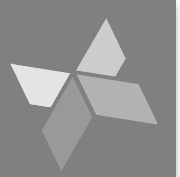
```
wadl2java -o diretorio
          -p pacote
          -s jaxrs20
          http://localhost:8080/festival/webapi/application.wadl
```

- Cria uma classe (**JAXBElement**) para **cada resource raiz**
- Cria uma classe **Servidor_WebctxRestctx.class** (para um servico em **http://servidor/webctx/restctx**). Ex: **Localhost_FestivalWebapi.class**
  - Esta classe possui um método **createClient()** que cria um cliente **JAX-RS 2.0** já configurado para usar o serviço
  - Também fornece métodos para retornar instâncias e representações dos objetos mapeados como resource



Exercícios

- 7. Escreva clientes REST (usando qualquer linguagem) que usem o serviço criado no exercício anterior para
  - **Preencher** o banco com 10 produtos
  - **Listar** os produtos disponíveis
  - **Remover** um produto
  - **Alterar** o preço de um produto
  - **Listar** produtos que custam mais de 1000
- 8. Exporte um WADL e escreva um cliente JAX-RS usando classes geradas



JAVAEE7

JAX-RS

RESTful web services

4

segurança

Autenticação HTTP

- A autenticação HTTP é realizada por **cabeçalhos**
- Exemplo (autenticação Basic)
  - 1. Resposta do servidor a tentativa de acesso a dados protegidos

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

  - 2. Requisição do cliente enviando credenciais

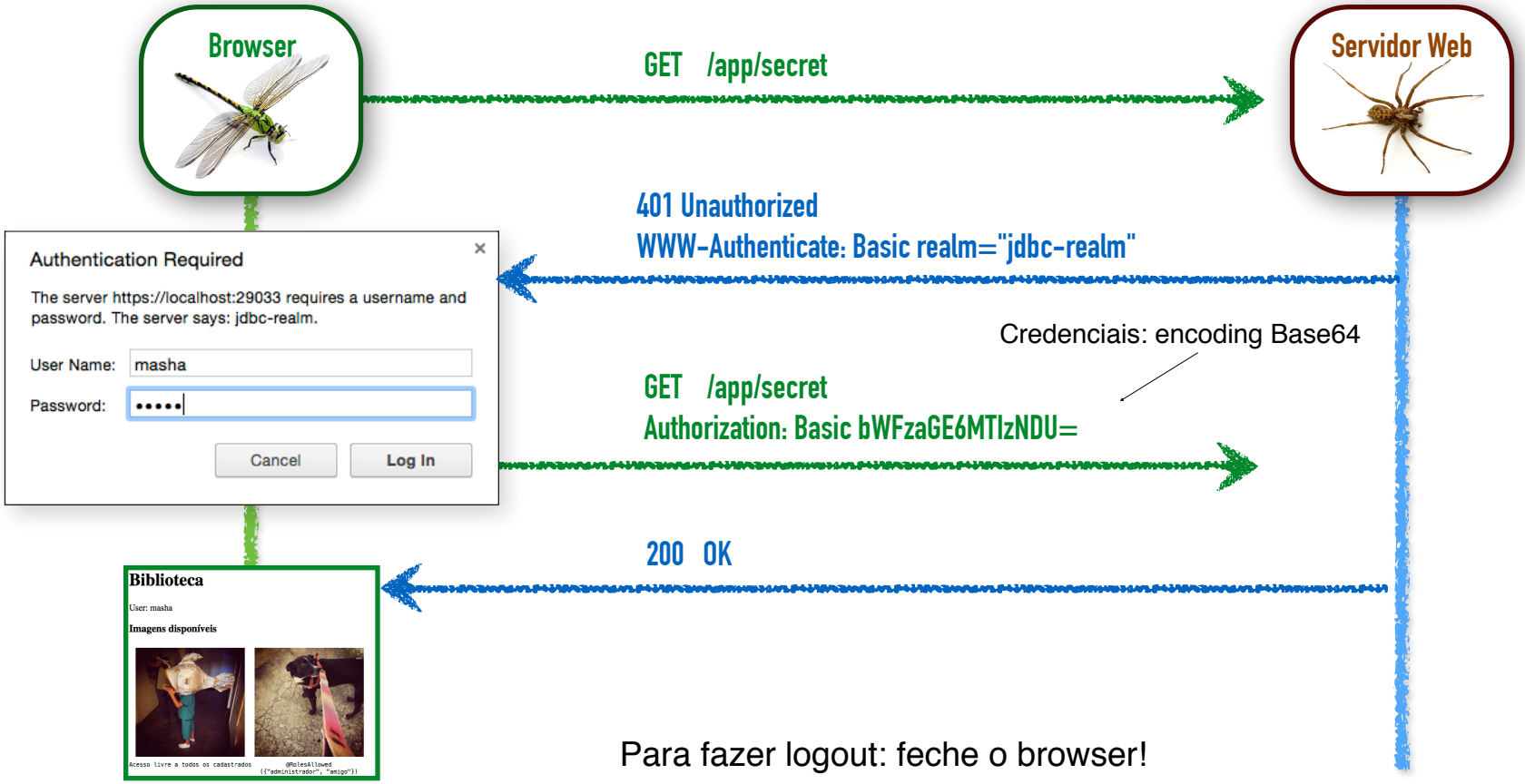
```
GET /ctx/app/filmes HTTP/1.1
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

  - 3. Se a autenticação falhar, o servidor responde com 403 Forbidden
- A configuração da autenticação deve ser realizada no servidor via **web.xml** (BASIC, DIGEST, CLIENT-CERT ou JASPIC)

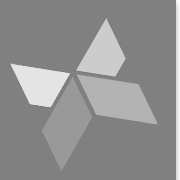
BASIC (RFC 2069 / 2617)

```
<login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>jdbc-realm</realm-name>
</login-config>
```

configuração
web.xml



Para fazer logout: feche o browser!



Autorização em web.xml

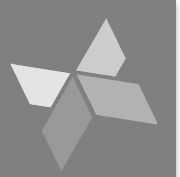
- É preciso declarar os roles no web.xml usando **<security-role>** (em servlets pode ser via anotações).
  - Mapeamento dos roles com usuários/grupos é dependente de servidor (glassfish-web.xml, jboss-web.xml, etc.)

```
<web-app> ...
        <security-role>
                <role-name>administrador</role-name>
        </security-role>

        <security-role>
                <role-name>amigo</role-name>
        </security-role>

        <security-role>
                <role-name>especial</role-name>
        </security-role>
   ...
</web-app>
```

Security constraints

- Bloco <security-constraint> contém três partes
  - **Um ou mais** <web-resource-collection>
  - **Pode** conter **um** <auth-constraint> (lista de roles)
  - **Pode** conter **um** <user-data-constraint> (SSL/TLS)

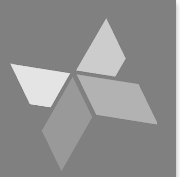
```
<security-constraint>
	<web-resource-collection>
		<web-resource-name>Área restrita</web-resource-name>
		<url-pattern>/secreto/*</url-pattern>
	</web-resource-collection>
	<auth-constraint>
		<role-name>administrador</role-name>
	</auth-constraint>
</security-constraint>
```



Web resource collection

- Agrupa recursos e operações controladas através de padrões de URL + métodos HTTP
  - Métodos HTTP não informados são restritos, mas métodos não informados estão descobertos
  - Use <deny-uncovered-http-methods/> ou outro bloco negando acesso a todos exceto <http-method-omission>

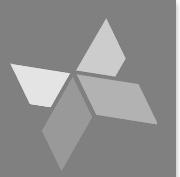
```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>Área restrita</web-resource-name>
                <url-pattern>/secreto/*</url-pattern>
                <url-pattern>/faces/castelo.xhtml</url-pattern>
                <http-method>GET</http-method>
                <http-method>POST</http-method>
        </web-resource-collection>
    ...
</security-constraint>
```



Authorization constraint

- Com **<auth-constraint>** as **<web-resource-collection>** são acessíveis apenas aos **<role-name>** declarados
  - Sem **<auth-constraint>**: sem controle de acesso (livre)
  - **<auth-constraint />** vazio: ninguém tem acesso

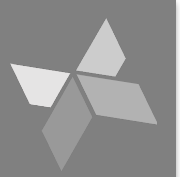
```
<web-app>
        ...
        <security-constraint>
                <web-resource-collection> ... </web-resource-collection>
                <web-resource-collection> ... </web-resource-collection>
                <auth-constraint>
                        <role-name>administrador</role-name>
                        <role-name>especial</role-name>
                </auth-constraint>
        </security-constraint>
                ...
        <security-constraint> ....</security-constraint>
</web-app>
```



Transport guarantee

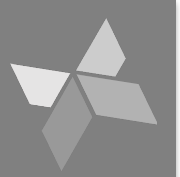
- Garantias mínimas para comunicação segura SSL/TLS
  - **NONE**: (ou ausente) não garante comunicação segura
  - **INTEGRAL**: proteção integral, autenticação no servidor
  - **CONFIDENTIAL**: proteção integral, autenticação: cliente e servidor

```
<web-app> ...
        <security-constraint>
                ...
                <user-data-constraint>
                        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
                </user-data-constraint>
        </security-constraint>
        <security-constraint>
                ...
                <user-data-constraint>
                        <transport-guarantee>INTEGRAL</transport-guarantee>
                </user-data-constraint>
        </security-constraint>
</web-app>
```



API de segurança em JAX-RS

- Métodos de **`javax.ws.rs.core.SecurityContext`**
- `getAuthenticationScheme()`
  - Retorna o string contendo o esquema de autenticação (mesmo que `HttpServletRequest#getAuthType()`)
- `isSecure()`
  - Retorna true se a requisição foi feita em canal seguro
- `getUserPrincipal()`
  - Retorna java.security.Principal com usuário autenticado
- `isUserInRole(String role)`
  - Retorna true se usuário autenticado pertence ao role



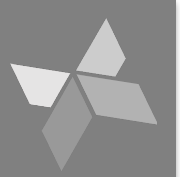
Exercícios

- 9. Configure o acesso à aplicação de forma que apenas usuários do grupo "**admin**" possam remover produtos ou alterar preços. Use autenticação BASIC.
- 10. Escreva um cliente HTTP comum (usando uma biblioteca como Apache HttpComponent) e envie cabeçalhos para autenticação BASIC.
- 11. Escreva um cliente Jersey que realize a autenticação e acesso.
- 12. Escreva um cliente JavaScript (JQuery/Angular) para autenticar e acessar o serviço

Exercícios de revisão

- Faça o exercício 12 da aplicação Biblioteca*:
  - 1. Cliente REST para obter as capas dos livros, título e assunto
  - 2. Exporta serviço de Autor como WS RESTful
  - 3. Testar acesso usando RESTClient e cliente Jersey/JAX-RS 2.0
  - 4. Cliente Angular acessando o serviço

*Exercicios: http://www.argonavis.com.br/download/exercicios_javaee.html



JAVAEE 7

Referências

- Especificações
  - HTTP: http://www.w3.org/Protocols/
  - RFC 2616 (HTTP) http://www.ietf.org/rfc/rfc2616.txt
  - WADL https://wadl.java.net/
  - Arquiteturas de Web Services http://www.w3.org/TR/ws-arch/

- Artigos, documentação e tutoriais
  - Saldhana. "Understanding Web Security using web.xml via Use Cases" Dzone. 2009. http://java.dzone.com/articles/understanding-web-security
  - Configuração de JAAS no Jboss 7 https://community.jboss.org/wiki/JBossAS7SecurityDomainModel
  - Java EE Tutorial sobre RESTful WebServices http://docs.oracle.com/javaee/7/tutorial/doc/jaxrs.htm
  - Documentação do Jersey (tutorial de JAX-RS): https://jersey.java.net/documentation/latest
  - 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
  - Alex Rodriguez. "RESTful Web Services: the basics". IBM Developerworks, 2008. http://www.ibm.com/developerworks/webservices/library/ws-restful/
  - Hadyel & Sandoz (editors). "JAX-RS: Java API for RESTful Web Services. Version 1.1. Oracle, 2009. https://jax-rs-spec.java.net/