

REST

Construa API's inteligentes
de maneira simples



Casa do
Código

ALEXANDRE SAUDATE

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

[2016]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

AGRADECIMENTOS

Muitas pessoas colaboraram direta ou indiretamente para a realização deste livro.

Gostaria de agradecer, primeiramente, à minha esposa e à minha família pela paciência e compreensão. Muitas vezes não pude sair de casa ou comparecer a compromissos porque estava ocupado com mais esta missão, e agradeço muito a compreensão de todos durante este período.

Também gostaria de agradecer ao Felipe Oliveira e à equipe da SOA|EXPERT, pelo aprendizado proporcionado e pela confiança no meu trabalho. Acredito que muitos não sabem, mas REST é apenas uma das facetas de SOA e é algo que eu leciono há quase dois anos na SOA|EXPERT. Obviamente, houve muito que aprendi neste processo, e sou realmente muito grato à SOA|EXPERT por isso.

Gostaria de agradecer também a várias pessoas com quem já conversei (ou ainda converso) sobre REST, e me ajudaram muito a entender os conceitos e práticas. Algumas dessas pessoas são o já citado Felipe Oliveira, da SOA|EXPERT, Bruno Pereira, da Rivendel, Luca Bastos, da Thoughtworks (que, aliás, me deu a honra de escrever o prefácio deste livro) e Victor Nascimento, da Concrete Solutions. Tenho certeza de que estou esquecendo de algumas pessoas, mas saibam que, de qualquer forma, sou muito grato a todos.

Também gostaria de agradecer ao Adriano Almeida e ao Paulo Silveira pela oportunidade e pela enorme paciência em me ajudar com a escrita deste livro.

Finalmente, gostaria de agradecer à comunidade de desenvolvimento como um todo. Sem o apoio desta, este livro com

certeza não seria possível.

PREFÁCIO

Este livro lotado de exemplos em Java é, na verdade, um guia de boas práticas para construir e compartilhar serviços REST na web. Serviços que escalem, que possam ser reutilizados e cuja manutenção não seja o mesmo pesadelo dos web services tradicionais.

Espero que meu resumo meio árido não desanime o leitor, porque o livro é muito bom. Na verdade, Alexandre Saudate consegue explicar as coisas de forma bem mais clara. A didática é o forte dele.

Este livro se destina a todos que querem conhecer mais sobre REST de uma forma prática. Um exercício interessante para o leitor seria experimentar refazer os muitos exemplos em Java, usando outras linguagens e frameworks.

Fiquei orgulhoso quando o Alexandre me convidou para escrever estas palavras de apresentação. Admiro sua disposição de compartilhar conhecimento desde os tempos do GUJ, onde suas respostas sempre foram generosas e esclarecedoras. Nisto temos completa identificação. Eu também sempre fui adepto de compartilhar aprendizados e aprender tentando ensinar os outros.

Não perguntei ao Alexandre o motivo de me ter convidado, mas tenho cá minhas suposições. Trabalhamos por um tempo juntos. Na época, já sabia da paixão dele por integração de aplicações, web services, SOA e afins. Ele estava escrevendo o primeiro livro e já era instrutor. E sabia que desde há muito compartilho dos mesmos interesses.

Já dei tutoriais sobre web services tradicionais, mas por algumas dores no meu caminhar, prefiro usar a alternativa REST sempre que

possível. Mesmo no caso de transações financeiras, caso que alguns dizem ser somente possível com os web services tradicionais.

Quando trabalhei junto com o Alexandre, volta e meia o assunto integração aparecia, dados os nossos interesses comuns. Eram conversas bem enriquecedoras em que ambos aprendíamos bastante.

Eu segui por outros caminhos mais generalistas e ele permaneceu fiel. Transformou sua paixão por integração de sistemas em livros. Este, seu segundo, usa Java nos exemplos, mas a maior parte do seu conteúdo pode ser seguido com facilidade, mesmo pelos que não conhecem Java a fundo. Os três primeiros capítulos formam a base teórica do livro.

Gostei muito de o Alexandre ter dedicado um capítulo inteiro ao protocolo HTTP. Em várias empresas pelas quais passei, adotei como missão pessoal convencer os desenvolvedores web da necessidade de conhecer HTTP. No caso do uso de REST, isto é praticamente indispensável.

Descontem o fato de eu ter sido apaixonado pelo tema quando digo que gostei muito de ler este livro. Mas de todo modo, acredito que muito mais gente gostará tanto quanto eu. Parabéns Alexandre. Nós estávamos precisando de um texto claro assim em português. E parabéns à Casa do Código, por propiciar oportunidade aos nossos especialistas de compartilhar conhecimento na nossa própria língua.

Por Luca Bastos

Luca Bastos é engenheiro, desenvolve sistemas desde 1968 e foi empreendedor por 17 anos vendendo serviços de TI. Hoje é consultor na ThoughtWorks.

INTRODUÇÃO

Estou certo de que você, leitor, ao comprar este livro, esperava descobrir o que é REST. REST é uma revolução. É um antagonismo ao *status quo* que já era previamente estabelecido antes de sua adoção em massa, ao estilo de RPC (*Remote Procedure Call*) e *web services* tradicionais.

E como nasceu, então? Oras, como toda revolução, nasce do desejo de mudar, de melhorar as coisas e de trazer contrapontos a pontos de vista preestabelecidos.

Este é um dos maiores desafios que este livro procura solucionar. Como instrutor do assunto na SOA|EXPERT, é sempre desafiador explicar a meus alunos a diferença entre modelos RPC tradicionais e a proposta de REST.

A proposta, claro, é mais do que oferecer uma simples *overview*; é dar ao leitor a noção dos princípios e teorias que regem REST, para depois apresentar casos reais de desenvolvimento.

Praticamente todos os exemplos são feitos em Java. No entanto, por ser uma linguagem executada na JVM, acredito que este modelo seja o mais abrangente possível. Se você não programa em Java, não se preocupe – como disse anteriormente, a ideia é apresentar não somente **como** fazer, mas **por quê**. Com o conhecimento que você adquirir lendo este livro, tenho certeza de que será fácil reproduzir os exemplos em outras linguagens.

Conteúdo do livro

Este livro possui dez capítulos.

O capítulo *Por que utilizar REST?* trata de dar uma visão rápida

de quais são os desafios que REST procura vencer, e como construir um primeiro serviço e um cliente REST. Ainda que bem básicos, os exemplos mostram algumas técnicas de desenvolvimento tanto no lado do cliente quanto no lado do servidor.

O capítulo *O protocolo HTTP* mostra a fundo o protocolo HTTP, que é a base sobre a qual todo serviço REST funciona. Este capítulo é um alicerce sobre o qual o restante do livro será baseado.

O capítulo *Conceitos de REST* vai utilizar os conceitos apresentados no capítulo anterior para entrar a fundo nas ideias por trás de REST.

O capítulo *Tipos de dados* vai dar mais informações a respeito de possíveis formatos de dados utilizados em transferência de informações usando REST. Este capítulo vai prover informações a respeito de estruturas de dados, validação geração destes dados.

No capítulo *Implementando serviços REST em Java com Servlets*, serão produzidos serviços utilizando *servlets Java*, ou seja, a API mais básica por trás de aplicações Java para *web*.

No capítulo *Explore o JAX-RS*, será apresentada a especificação Java para construção de serviços, a JAX-RS. Abordo neste livro a versão 2 desta API, que acrescenta uma série de novas facilidades à versão anterior.

No capítulo *REST, client-side*, serão apresentados modelos de construção de clientes REST. Serão mostrados como desenvolver estes clientes usando a API padrão da JAX-RS 2 com a implementação de referência (o Jersey), o framework RESTEasy e JavaScript puro.

No capítulo *Segurança REST*, falarei sobre sistemas de segurança aplicados a REST. Estão compreendidos proteção com HTTPS, autenticação Basic e Digest e, então, autenticação com o

protocolo OAuth.

No capítulo *Tópicos avançados de serviços REST*, serão apresentados tópicos avançados de desenvolvimento em REST. Como desenvolver serviços que passam por acessos simultâneos? Como transformar complexas funções baseadas em RPC para o modelo REST? Como realizar testes automatizados destes serviços?

Finalmente, o capítulo *Perguntas frequentes sobre REST* apresenta uma seção de dúvidas frequentes. Posso utilizar REST em larga escala? Quando devo utilizar REST em vez de WS-* (e vice-versa)?

Com esta disposição de capítulos, espero que os leitores consigam orientar seu próprio aprendizado. Se você já trabalha com REST em Java, e está apenas procurando maneiras de solucionar certos problemas corriqueiros, sugiro realizar a leitura do capítulo *Segurança REST* em diante.

Caso você já trabalhe com REST, mas não em Java, e quer migrar para esta linguagem, leia do capítulo *Implementando serviços REST em Java com Servlets* em diante.

Se você não trabalha com REST, mas não quer adotar a plataforma Java, sugiro descartar a leitura apenas dos capítulos *Implementando serviços REST em Java com Servlets* e *Explore o JAX-RS*.

Caso você não trabalhe com REST e quer adotar a plataforma Java para seus exercícios, leia **todos** os capítulos.

Recursos do livro

Caso você tenha quaisquer dúvidas a respeito do conteúdo deste, acesse o repositório de código-fonte do livro: <https://github.com/alesaudate/rest>.

Você também pode tirar suas dúvidas e entrar em contato comigo no fórum oficial: <http://forum.casadocodigo.com.br/>.

Boa leitura!

Sumário

1 Por que utilizar REST?	1
1.1 HTML tradicional versus REST	1
1.2 Web services	2
1.3 REST	3
1.4 Como elaborar a URL	6
1.5 Desenvolvendo um protótipo de web service REST	7
1.6 Avançando o protótipo de web service REST	10
1.7 Conclusão	12
2 O protocolo HTTP	13
2.1 Os fundamentos do HTTP	13
2.2 Métodos HTTP	15
2.3 Idempotência: o efeito de sucessivas invocações	16
2.4 Segurança dos métodos	17
2.5 Tipos de passagem de parâmetros	18
2.6 Cabeçalhos HTTP	22
2.7 Media types	23
2.8 Códigos de status	27
2.9 Conclusão	30
3 Conceitos de REST	32

3.1 Semânticas de recursos	32
3.2 Interação por métodos	34
3.3 Representações distintas	36
3.4 Uso correto de status codes	37
3.5 HATEOAS	39
3.6 Boas práticas de passagem de parâmetros	44
3.7 Conclusão	46
4 Tipos de dados	47
4.1 Aprenda a lidar com XML	47
4.2 Ferramental XML: conhecendo os XML Schemas	48
4.3 Trabalhando com XML utilizando JAXB	59
4.4 Testando o XML gerado	65
4.5 Utilizando JAXB sem um XML Schema	66
4.6 JSON	68
4.7 Trabalhando com JSON utilizando JAXB	70
4.8 Validação de JSON com JSON Schema	74
4.9 Conclusão	79
5 Implementando serviços REST em Java com Servlets	81
5.1 Uma implementação com Servlets	81
5.2 Implementando negociação de conteúdo	94
5.3 Implementando a busca por uma cerveja específica	97
5.4 Implementando a criação de um recurso	102
5.5 Implementando negociação de conteúdo no recurso	108
5.6 Conclusão	112
6 Explore o JAX-RS	114
6.1 Configurando o JAX-RS	114
6.2 Criando o primeiro serviço JAX-RS	116
6.3 O que aconteceu?	118

6.4 Desenvolvendo os métodos de consulta de cervejas	119
6.5 Implementando o método de criação de novas cervejas	124
6.6 Métodos para atualizar e apagar recursos	128
6.7 Implementando links HATEOAS	129
6.8 Implementando paginação na consulta	132
6.9 Adaptando o código para retornar JSON	135
6.10 Implementando retorno de dados binários	136
6.11 Implementando consumo de dados binários	138
6.12 Conclusão	141
7 REST, client-side	142
7.1 JavaScript	142
7.2 JAX-RS	157
7.3 Implementação dos clientes com o RESTEasy	166
7.4 WADL	169
7.5 Conclusão	178
8 Segurança REST	180
8.1 Segurança contra o quê?	180
8.2 Proteção contra interceptação com HTTPS	182
8.3 Implantando SSL no nosso servidor	187
8.4 HTTP Basic	189
8.5 Modificando o cliente	193
8.6 HTTP Digest	195
8.7 OAuth	198
8.8 Implementando acesso ao Twitter através do Jersey	202
8.9 Construindo um servidor OAuth	224
8.10 Conclusão	231
9 Tópicos avançados de serviços REST	233
9.1 Busca por exemplos	233

9.2 Transformação de funções em REST — validação de CPF/CNPJ	238
9.3 Transformação de funções em REST — envio de e-mails	241
9.4 Serviços assíncronos	250
9.5 Atualizações concorrentes	256
9.6 Cacheamento de resultados	259
9.7 Testes automatizados de serviços REST	267
9.8 Criando parsers personalizados de dados	272
9.9 Conclusão	279
10 Perguntas frequentes sobre REST	280
10.1 É possível usar REST em larga escala?	280
10.2 Como gerenciar múltiplas versões de serviços?	281
10.3 Quantos e quais links HATEOAS devo inserir nos meus recursos?	282
10.4 Quando eu devo preferir utilizar REST em vez de WS-*?	283
10.5 Quando eu devo preferir utilizar REST em vez de comunicação nativa da minha linguagem de programação?	284
11 Conclusão	286
12 Referências bibliográficas	288

Versão: 19.5.7

CAPÍTULO 1

POR QUE UTILIZAR REST?

"Não é merecedor do favo de mel aquele que evita a colmeia porque as abelhas têm ferrões." – William Shakespeare

Você é um empreendedor. Seguindo a filosofia de uma *startup*, você quer produzir um *site* para seu produto, uma loja de cervejas finas pela internet. Assim, você quer fazer um sistema que seja rápido de produzir e, ao mesmo tempo, seja eficiente e fácil de reutilizar em ambientes diferentes (você está considerando uma versão *mobile* para o sistema). Logo, você explora diversas possibilidades.

1.1 HTML TRADICIONAL VERSUS REST

Você decide avaliar, antes de tudo, sistemas tradicionais web baseados em formulários HTML.

Fica claro para você que fazer um sistema tradicional, baseado em formulários HTML, está fora de questão. Isso porque esse tipo de formulário é claramente otimizado para trabalhar com sistemas baseados na web, mas você não tem certeza de quanto ele será flexível para trabalhar com outros tipos de *front-end*. Por exemplo, considere uma tela para cadastro dos clientes do site:

```
<html>
<body>
  <form action="/cadastrar" method="post">
    <input type="text" name="nome" />
    <input type="text" name="dataNascimento" />
```

```
        <input type="submit" value="Cadastrar" />
    </form>
</body>
</html>
```

Este código (com um pouco mais de tratamento) deve atendê-lo bem. No entanto, após submeter o formulário, o servidor deve trazer uma página HTML – o que, obviamente, não é desejável em um cenário com vários front-ends, como um aplicativo para Android ou iOS.

Sua próxima possibilidade é, portanto, avaliar outros tipos de sistemas.

1.2 WEB SERVICES

O próximo cenário que vem à sua mente é o de uso de *web services* tradicionais, ou seja, baseados em SOAP. A princípio, parece uma boa ideia, já que o mecanismo de tráfego de informações é via XML e, portanto, é possível interagir com serviços desse tipo a partir de código JavaScript (ou seja, possibilitando a interação tanto a partir de um *browser* quanto a partir de código puro).

Porém, estudando mais, você percebe que interagir com este tipo de web service não é tão simples quanto você gostaria. Por exemplo, para listar os clientes do seu sistema, é necessário enviar o seguinte XML para o servidor:

```
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <listarClientes
            xmlns="http://geladaonline.com.br/administracao/1.0/
                service" />
    </soap:Body>
</soap:Envelope>
```

Ao que você receberá como resposta:

```
<soap:Envelope xmlns:domain=
    "http://geladaonline.com.br/administracao/1.0/domain"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
    <listarClientesResponse
        xmlns="http://geladaonline.com.br/administracao/1.0/
            service">
        <domain:clientes>
            <domain:cliente domain:id="1">
                <domain:nome>Alexandre</domain:nome>
                <domain:dataNascimento>
                    2012-12-01
                </domain:dataNascimento>
            </domain:cliente>
            <domain:cliente domain:id="2">
                <domain:nome>Paulo</domain:nome>
                <domain:dataNascimento>
                    2012-11-01
                </domain:dataNascimento>
            </domain:cliente>
        </domain:clientes>
    </listarClientesResponse>
</soap:Body>
</soap:Envelope>
```

Este código parece complicado demais para uma simples requisição de listagem de clientes. A complexidade (e verbosidade) do protocolo fica especialmente evidente quando você pensa no caso da comunicação feita com plataformas móveis, onde a comunicação pela rede deve ser a mais sucinta possível.

1.3 REST

Você estuda mais a respeito de web services quando você lê sobre serviços REST. Ao estudar sobre o assunto, parece-lhe uma técnica simples. Por exemplo, para realizar a listagem de clientes do seu sistema, você pode utilizar o navegador (seja este o Chrome, Firefox, Internet Explorer ou qualquer outro) para abrir a URL <http://localhost:8080/cervejaria/clientes>. Ao fazer isso, você pode obter o seguinte resultado:

```
<clientes>
```

```
<cliente id="1">
  <nome>Alexandre</nome>
  <dataNascimento>2012-12-01</dataNascimento>
</cliente>
<cliente id="2">
  <nome>Paulo</nome>
  <dataNascimento>2012-11-01</dataNascimento>
</cliente>
</clientes>
```

Mas, afinal, como fazer seu código interagir com esta listagem?
O que aconteceu?

Os princípios básicos de REST

REST significa *REpresentational State Transfer* (ou **Transferência de Estado Representativo**, em tradução livre), e é um estilo de desenvolvimento de web services que teve origem na tese de doutorado de Roy Fielding (2000). Este, por sua vez, é coautor de um dos protocolos mais usados no mundo, o HTTP (*HyperText Transfer Protocol*). Assim, é notável que o protocolo REST é guiado (dentre outros preceitos) pelo que seriam as boas práticas de uso de HTTP:

- Uso adequado dos métodos HTTP;
- Uso adequado de URLs;
- Uso de códigos de status padronizados para representação de sucessos ou falhas;
- Uso adequado de cabeçalhos HTTP;
- Interligações entre vários recursos diferentes.

O propósito deste livro é, portanto, exemplificar o que são estas práticas, guiar você, leitor, por meio delas e fornecer os insumos de como construir seus próprios serviços, de maneira que estes sejam sempre tão escaláveis, reutilizáveis e manuteníveis quanto possível.

As fundações de REST

O "marco zero" de REST é o **recurso**. Em REST, tudo é definido em termos de recursos, sendo estes os conjuntos de dados que são trafegados pelo protocolo. Os recursos são representados por URIs. Note que, na web, URIs e URLs são essencialmente a mesma coisa – razão pela qual vou usar os dois termos neste livro de forma intercalada.

QUAL A DIFERENÇA ENTRE UMA URL E UMA URI?

URL significa *Universal Resource Locator* e URI, *Universal Resource Identifier*. Uma URI, como diz o próprio nome, pode ser utilizada para identificar qualquer coisa – dar um caminho para um determinado conteúdo, dar nome a este etc. (BERNERS-LEE; FIELDING; MASINTER, 2005).

Já uma URL pode ser usada apenas para fornecer caminhos – sendo que uma URL é, portanto, uma forma de uma URI. É mais natural que URIs que não sejam URLs sejam utilizadas em outros contextos, como fornecimento de *namespaces* XML.

Tomando como exemplo o caso da listagem de clientes, é possível decompor a URL usada para localização da listagem em várias partes. Veja <http://localhost:8080/cervejaria/clientes> como exemplo:

- **http://**: indica o protocolo que está sendo utilizado (no caso, HTTP).
- **localhost:8080**: indica o servidor de rede que está sendo usado e a porta (quando a porta não é especificada, assume-se que é a padrão – no caso do protocolo HTTP, 80).
- **cervejaria**: indica o **contexto** da aplicação, ou seja, a

raiz pela qual a aplicação está sendo fornecida para o cliente. Vou me referir a esta, daqui em diante, como **contexto da aplicação** ou apenas **contexto**.

- **clientes**: é o endereço, de fato, do recurso – no caso, a listagem de clientes. Vou me referir a este, daqui em diante, como **endereço do recurso**.

O protocolo

O protocolo, em realidade, não é uma restrição em REST – em teoria. Na prática, o HTTP é o único protocolo 100% compatível conhecido. Vale destacar que o HTTPS (*HyperText Transfer Protocol over Secure Sockets Layer*) não é uma variação do HTTP, mas apenas a adição de uma camada extra – o que mantém o HTTPS na mesma categoria que o HTTP, assim como qualquer outro protocolo que seja utilizado sobre o HTTP, como SPDY (PEON; BELSHE, 2012).

As possíveis causas para esta "preferência" podem ser apontadas: o autor do HTTP também é o autor de REST e, além disso, o protocolo HTTP é um dos mais usados no mundo (sendo que a web, de maneira geral, é fornecida por este protocolo). Estes fatos promoveriam uma aceitação grande e rápida absorção de REST pela comunidade de desenvolvedores. No capítulo *O protocolo HTTP*, veremos mais a respeito disso.

1.4 COMO ELABORAR A URL

A URL escolhida deve ser única por recurso. Isto significa que, sempre que desejar obter a representação de todos os clientes, você deve utilizar a URL <http://localhost:8080/cervejaria/clientes>. Realizar uma consulta nesta URL pode retornar dados da seguinte maneira:

```
<clientes>
  <cliente id="1">
    <nome>Alexandre</nome>
    <dataNascimento>2012-12-01</dataNascimento>
  </cliente>
  <cliente id="2">
    <nome>Paulo</nome>
    <dataNascimento>2012-11-01</dataNascimento>
  </cliente>
</clientes>
```

Se você desejar, portanto, retornar um cliente específico, você deve usar uma URL diferente. Suponha, por exemplo, que você deseja retornar o cliente com `id` igual a 1. Neste caso, a URL seria <http://localhost:8080/cervejaria/clientes/1>. Isso retornaria algo como:

```
<cliente id="1">
  <nome>Alexandre</nome>
  <dataNascimento>2012-12-01</dataNascimento>
</cliente>
```

1.5 DESENVOLVENDO UM PROTÓTIPO DE WEB SERVICE REST

Para desenvolver um protótipo de web service REST bem simples, não é necessário utilizar-se de alta tecnologia. Basta que o desenvolvedor tenha em mente o resultado que deseja alcançar e criar um arquivo contendo a resposta desejada. Por exemplo, suponha um arquivo `clientes.xml`, com o conteúdo:

```
<clientes>
  <cliente id="1">
    <nome>Alexandre</nome>
    <dataNascimento>2012-12-01</dataNascimento>
  </cliente>
  <cliente id="2">
    <nome>Paulo</nome>
    <dataNascimento>2012-11-01</dataNascimento>
  </cliente>
</clientes>
```

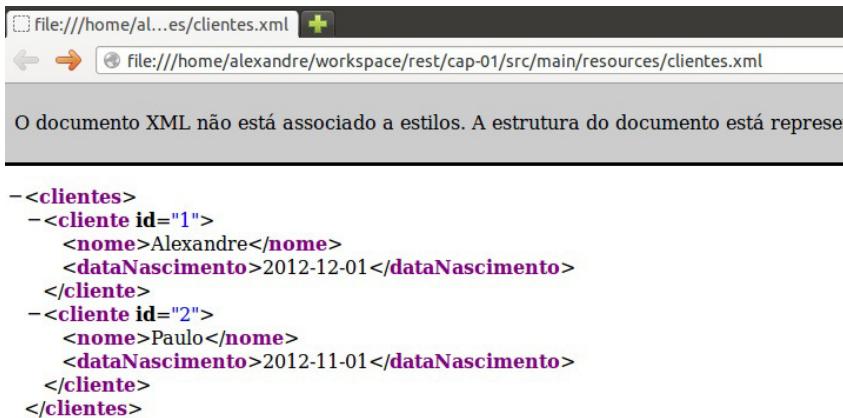
Para acessá-lo pelo browser, basta utilizar a URL `file://<caminho do arquivo>`. Por exemplo, no meu caso (ambiente Linux), o arquivo está em `/home/alexandre/workspace/rest/cap-01/src/main/resources/ clientes.xml`.

Assim, a URL completa fica `file:///home/alexandre/workspace/rest/cap-01/src/main/resources/clientes.xml`. Note que são três barras no começo (duas para delimitação do protocolo e outra para indicar a raiz do sistema Linux).

URL PARA ARQUIVOS NO WINDOWS

No Windows, para renderizar uma URL, também é necessário utilizar três barras no começo. Por exemplo, para localizar um arquivo em Documentos , o caminho a ser inserido no browser ficaria similar a `file:///C:/Users/Alexandre/Documents/clientes.xml` . Isto é devido a uma particularidade do próprio sistema de arquivos do Windows, que pode ter várias raízes (C: , D: , e assim por diante).

Desta forma, o conteúdo é renderizado pelo browser:



The screenshot shows a browser window with the URL 'file:///home/alexandre/workspace/rest/cap-01/src/main/resources/clientes.xml'. The page content displays the XML structure:

```
-<clientes>
-<cliente id="1">
    <nome>Alexandre</nome>
    <dataNascimento>2012-12-01</dataNascimento>
</cliente>
-<cliente id="2">
    <nome>Paulo</nome>
    <dataNascimento>2012-11-01</dataNascimento>
</cliente>
</clientes>
```

Figura 1.1: XML renderizado pelo browser

Para desenvolver um cliente para este arquivo, basta usar a API de I/O do próprio Java, começando pela classe `java.net.URL`, que indica qual o caminho a ser seguido:

```
String caminho = "file:///home/alexandre/workspace/rest/" +
                 "cap-01/src/main/resources/clientes.xml";
URL url = new URL(caminho);
```

Feito isso, o próximo passo é conectar-se a este caminho e abrir uma *input stream*, ou seja, um canal de leitura de dados do caminho indicado pela URL. Para conectar-se, basta utilizar o método `openConnection` da URL e, para abrir a *input stream*, basta usar o método `getInputStream`:

```
java.io.InputStream inputStream =
    url.openConnection().getInputStream();
```

O próximo passo é encapsular esta leitura em um `java.io.BufferedReader`. Esta classe possui um método utilitário para ler informações linha a linha, facilitando o processo de leitura de dados. Para utilizá-lo, no entanto, é necessário encapsular a `InputStream` em um

```
java.io.InputStreamReader :
```

```
BufferedReader bufferedReader = new BufferedReader(  
    new InputStreamReader(inputStream));
```

Uma vez feito esse procedimento, basta usar o método `readLine` em um laço, até que este método retorne `null`:

```
String line = null;  
  
while ((line = bufferedReader.readLine()) != null) {  
    System.out.println(line);  
}
```

O código completo fica assim:

```
package br.com.cervejaria.cliente;  
  
import java.io.*;  
import java.net.URL;  
  
public class Cliente {  
  
    public static void main(String[] args) throws IOException {  
        String caminho = "file:///home/alexandre/workspace/rest/"  
            + "cap-01/src/main/resources/clientes.xml";  
        URL url = new URL(caminho);  
        InputStream inputStream =  
            url.openConnection().getInputStream();  
        BufferedReader bufferedReader = new BufferedReader(  
            new InputStreamReader(inputStream));  
  
        String line = null;  
  
        while ((line = bufferedReader.readLine()) != null) {  
            System.out.println(line);  
        }  
    }  
}
```

1.6 AVANÇANDO O PROTÓTIPO DE WEB SERVICE REST

Para desenvolver um protótipo que seja verdadeiramente

baseado na web, basta um simples *servlet*. No código-fonte deste livro, estou utilizando a especificação 3.0 de *servlets*, que me permite anotá-los para que sejam detectados pelo contêiner.

Para realizar a listagem de clientes, basta fazer com que o servlet leia o arquivo `clientes.xml`, usado anteriormente. Assim, o servlet será capaz de fornecer estas informações para o cliente. Este fornecimento de dados vai ser feito via método `GET` (mais à frente, no capítulo *O protocolo HTTP*, você vai entender o motivo).

O código do servlet vai ficar assim:

```
package br.com.cervejaria.servlet;

import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet(urlPatterns = "/clientes", loadOnStartup = 1)
public class ClientesServlet extends HttpServlet {

    private String clientes;

    @Override
    protected void doGet(HttpServletRequest req,
                          HttpServletResponse resp)
        throws ServletException, IOException{
        resp.getWriter().print(clientes);
    }

    @Override
    public void init() throws ServletException {
        // Faz aqui a leitura do arquivo clientes.xml
        // E coloca na String clientes
    }
}
```

O CÓDIGO-FONTE DO LIVRO

No código-fonte do livro (disponível em <https://github.com/alesaudate/rest/>), você encontra um projeto Maven com este código pronto para ser inicializado. Basta executar `mvn jetty:run`.

Ao acessar a URL <http://localhost:8080/cervejaria/clientes>, você obtém como resposta exatamente o conteúdo do arquivo. E para realizar isto com código, basta alterar o código-fonte do cliente criado anteriormente para esta mesma URL:

```
URL url = new URL("http://localhost:8080/cervejaria/clientes");
```

1.7 CONCLUSÃO

Neste capítulo, você pôde conferir alguns dos principais motivos pelos quais o modelo REST é selecionado para se trabalhar em aplicações modernas. Você conferiu alguns dos princípios mais básicos de REST, como o uso de URLs e como realizar o acesso a estes documentos através de browsers e código Java.

Observe que este capítulo é apenas uma ideia do que está por vir. Ainda há muito a seguir e muito a ser melhorado no nosso código, a começar pelos princípios do protocolo HTTP (nos quais REST é fortemente baseado) e entender como criar serviços mais complexos utilizando este paradigma.

CAPÍTULO 2

O PROTOCOLO HTTP

"Se vi mais longe, foi por estar de pé sobre ombros de gigantes." – Isaac Newton

Como mencionado anteriormente, o modelo REST foi desenvolvido por Roy Fielding, um dos criadores do protocolo HTTP. Portanto, fica naturalmente evidente, para quem conhece ambos os modelos, as semelhanças entre eles – sendo que, na realidade, REST é idealmente concebido para uso com o protocolo HTTP. Portanto, para ser um bom usuário de REST, é necessário ter um bom conhecimento do protocolo HTTP.

2.1 OS FUNDAMENTOS DO HTTP

O protocolo HTTP (*HyperText Transfer Protocol* – Protocolo de Transferência de Hipertexto) data de 1996, época em que os trabalhos conjuntos de Tim Berners-Lee, Roy Fielding e Henrik Frystyk Nielsen (1996) levaram à publicação de uma RFC (*Request for Comments*), descrevendo este protocolo. Trata-se de um protocolo de camada de aplicação, segundo o modelo OSI (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 1994) e, portanto, de relativa facilidade de manipulação em aplicações.

Este protocolo foi desenvolvido de maneira a ser o mais flexível possível para comportar diversas necessidades diferentes. Em linhas gerais, ele segue o seguinte formato de requisições:

```
<método> <URL> HTTP/<versão>
<Cabeçalhos - Sempre vários, um em cada linha>

<corpo da requisição>
```

Por exemplo, para realizar a requisição do capítulo passado, foi enviado para o servidor algo semelhante ao seguinte:

```
GET /cervejaria/clientes HTTP/1.1
Host: localhost:8080
Accept: text/xml
```

Portanto, o que foi passado nesta requisição foi:

```
GET /cervejaria/clientes HTTP/1.1
```

Com isso, o método `GET` foi utilizado para solicitar o conteúdo da URL `/cervejaria/clientes`. Além disso, o protocolo HTTP versão 1.1 foi usado.

Note que o *host*, ou seja, o servidor responsável por fornecer estes dados, é passado em um **cabeçalho** à parte. No caso, o *host* escolhido foi `localhost`, na porta `8080`. Além disso, um outro cabeçalho, `Accept`, foi fornecido. Este tem a função de informar ao servidor qual o tipo de informação será aceita.

A resposta para as requisições seguem o seguinte formato geral:

```
HTTP/<versão> <código de status> <descrição do código>
<cabeçalhos>

<resposta>
```

Por exemplo, a resposta para esta requisição foi semelhante à seguinte:

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 245

<clientes>
  <cliente id="1">
    <nome>Alexandre</nome>
```

```
<dataNascimento>2012-12-01</dataNascimento>
</cliente>
<cliente id="2">
    <nome>Paulo</nome>
    <dataNascimento>2012-11-01</dataNascimento>
</cliente>
</clientes>
```

Aqui, note que o código 200 indicou que a requisição foi bem-sucedida, e o cabeçalho `Content-Length` trouxe o tamanho da resposta – no caso, o XML que contém a listagem de clientes. Além disso, o cabeçalho `Content-Type` indica que a resposta é, de fato, XML – por meio de um tipo conhecido como `Media Type`.

2.2 MÉTODOS HTTP

A requisição que fizemos com o nosso navegador foi feita através do método `GET`. Sempre que utilizamos nosso navegador para enviar requisições, este é o método a ser usado, mas podemos usar mais maneiras. A versão corrente do HTTP, 1.1, define oficialmente nove métodos – embora o protocolo seja extensível em relação a estes.

Hoje, estes nove são:

- GET
- POST
- PUT
- DELETE
- OPTIONS
- HEAD
- TRACE
- CONNECT
- PATCH

Cada método possui particularidades e aplicações de acordo com a necessidade. Essas particularidades são definidas em termos

do efeito provocado por sucessivas invocações (**idempotência**), **segurança e mecanismo de passagem de parâmetros**. Além disso, cada um possui suas próprias particularidades de uso, que veremos mais adiante.

2.3 IDEMPOTÊNCIA: O EFEITO DE SUCESSIVAS INVOCACÕES

A idempotência de um método é relativa às modificações que são realizadas em informações do lado do servidor. Trata-se do efeito que uma mesma requisição tem do lado do servidor – se a mesma requisição, realizada múltiplas vezes, provoca alterações no lado do servidor como se fosse uma única, então esta é considerada idempotente.

Por exemplo, considere as quatro operações de bancos de dados: `SELECT` , `INSERT` , `UPDATE` e `DELETE` . Realizando um paralelo destas com o conceito de idempotência, observe o seguinte:

```
SELECT * from CLIENTES;
```

Note que esta requisição, para um banco de dados, terá o mesmo efeito todas as vezes que for executada (obviamente, assumindo que ninguém está fazendo alterações nas informações que já estavam gravadas).

Agora, observe o seguinte:

```
INSERT INTO CLIENTES VALUES (1, 'Alexandre');
```

Esta requisição, por sua vez, provocará diferentes efeitos sobre os dados do banco de dados todas as vezes que for executada, dado que está aumentando o tamanho da tabela. Portanto, ela não é considerada idempotente.

Note que este conceito não está relacionado à realização ou não

de modificações. Por exemplo, considere as operações `UPDATE` e `DELETE` :

```
UPDATE CLIENTES SET NOME = 'Paulo' WHERE ID = 1;
```

```
DELETE FROM CLIENTES WHERE ID = 1;
```

Note que, em ambos os casos, as alterações realizadas são idênticas a todas as subsequentes. Por exemplo, suponha os dados:

ID	NOME
1	Alexandre

Se a requisição de atualização for enviada, estes dados ficarão assim:

ID	NOME
1	Paulo

E então, caso a mesma requisição seja enviada repetidas vezes, ainda assim provocará o mesmo efeito que da primeira vez, sendo considerada, portanto, idempotente.

2.4 SEGURANÇA DOS MÉTODOS

Quanto à **segurança**, os métodos são assim considerados se não provocarem quaisquer alterações nos dados contidos. Ainda considerando o exemplo das operações de bancos de dados, por exemplo, o método `SELECT` pode ser considerado seguro – `INSERT` , `UPDATE` e `DELETE` , não.

Em relação a estas duas características, a seguinte distribuição dos métodos HTTP é feita:

	Idempotente	Seguro
GET	X	X
POST		
PUT	X	
DELETE	X	
OPTIONS	X	X
HEAD	X	X
PATCH	X	

Figura 2.1: Métodos HTTP, de acordo com idempotência e segurança

2.5 TIPOS DE PASSAGEM DE PARÂMETROS

Muitas vezes, é necessário passar parâmetros para URLs. Estes parâmetros podem conter a descrição de um determinado recurso, modificações de como exibir um recurso etc. Por exemplo, se quisermos recuperar todas as cervejas do tipo LAGER , podemos utilizar uma URL como a seguinte:

/cervejaria/cervejas?tipo=LAGER

Os métodos HTTP suportam parâmetros sob duas formas: os chamados *query parameters* e *body parameters*.

Os *query parameters* são passados na própria URL da requisição. Por exemplo, considere a seguinte requisição para o serviço de busca do Google:

<http://www.google.com.br/?q=HTTP>

Esta requisição faz com que o servidor do Google automaticamente entenda a string HTTP como um parâmetro. Inserir esta URL no browser automaticamente indica para o servidor que uma busca por HTTP está sendo realizada.

Os *query parameters* são inseridos a partir do sinal de interrogação. Este sinal indica para o protocolo HTTP que, dali em diante, serão usados *query parameters*. Esses são inseridos com formato = (assim como no exemplo, em que q é a chave e HTTP , o valor).

Caso mais de um parâmetro seja necessário, os pares são separados com um & . Por exemplo, a requisição para o serviço do Google poderia, também, ser enviada da seguinte maneira:

```
http://www.google.com.br/?q=HTTP&oq=HTTP
```

Os *query parameters* são enviados na própria URL. Isto quer dizer que a requisição para o Google é semelhante à seguinte:

```
GET /?q=HTTP&oq=HTTP HTTP/1.1  
Host: www.google.com.br
```

COMO CHECAR O TRÁFEGO VIA HTTP

Para visualizar o tráfego destes dados, várias técnicas estão à disposição. As mais simples são a instalação de um *plugin* do Firefox chamado Live HTTP Headers , ou apertar F12 no Chrome.

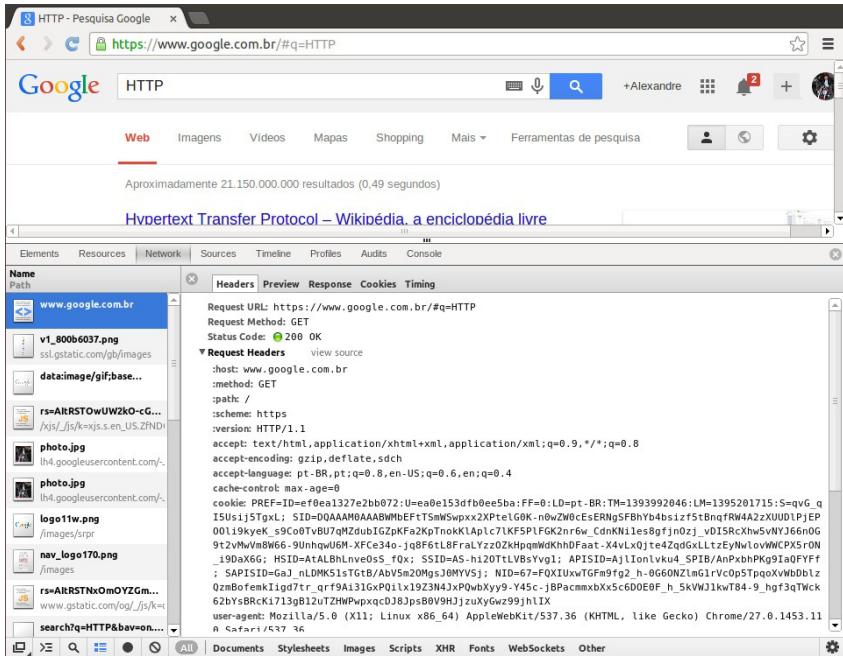


Figura 2.2: Checando o tráfego via HTTP

Note que o caractere de espaço é o separador entre o método HTTP, a URL e a versão do HTTP a ser utilizada. Desta forma, se for necessário usar espaços nos *query parameters*, é preciso fazer uso de uma técnica chamada **codificação da URL**. Esta técnica adapta as URLs para que elas sejam compatíveis com o mecanismo de envio de dados, e codificam não apenas espaços como outros caracteres especiais e acentuados.

Esta codificação segue a seguinte regra:

- Espaços podem ser codificados utilizando + ou a string %20;
- Letras maiúsculas e minúsculas, números e os caracteres ., -, ~ e _ são deixados como estão;
- Caracteres restantes são codificados de acordo com sua representação ASCII / UTF-8, e codificados como

hexadecimal.

Assim, a *string* **às vezes** é codificada como **%C3%A0s+vezes** (sendo que os *bytes* C3 e A0 representam os números 195 e 160 que, em codificação UTF-8, tornam-se a letra à).

Note que uma limitação dos *query parameters* é a impossibilidade de passar dados estruturados como parâmetro. Por exemplo, não há a capacidade de relacionar um *query param* com outro, levando o desenvolvedor a criar maneiras de contornar esta limitação. Suponha que seja necessário desenvolver uma pesquisa de clientes baseada em vários tipos de impostos devidos em um certo período de tempo. Esta pesquisa deveria fornecer:

- O nome do imposto;
- O período (data inicial e data final da busca).

Se esta pesquisa for baseada em um único imposto, não há problema algum. Mas tome como base uma pesquisa baseada como uma lista de impostos. Ela seria semelhante a:

```
/pessoas?imposto.1=IR&data.inicio.1=2011-01-01&data.inicio.2=
2012-01-01
```

Onde a numeração, neste caso, seria o elemento agrupador dos dados – ou seja, um *workaround* para a limitação de dados estruturados.

Quando há a necessidade de fornecer dados complexos, no entanto, é possível fazê-lo pelo corpo da requisição. Algo como:

```
POST /clientes HTTP/1.1
Host: localhost:8080
Content-Type: text/xml
Content-Length: 93

<cliente>
  <nome>Alexandre</nome>
  <dataNascimento>2012-01-01</dataNascimento>
```

```
</cliente>
```

Quanto a este tipo de parâmetro, não há limitações. O servidor faz a interpretação dos dados a partir do fato de que esses parâmetros são os últimos do documento, e controla o tamanho da leitura utilizando o cabeçalho `Content-Length`.

2.6 CABEÇALHOS HTTP

Os cabeçalhos em HTTP são usados para trafegar todo o tipo de metainformação a respeito das requisições. Vários deles são padronizados; no entanto, eles são facilmente extensíveis para comportar qualquer particularidade que uma aplicação possa requerer nesse sentido.

Por exemplo, ao realizar uma requisição, pelo Firefox, para o site <http://www.casadocodigo.com.br/>, as seguintes informações são enviadas:

```
GET / HTTP/1.1
Host: www.casadocodigo.com.br
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:19.0)
Gecko/20100101 Firefox/19.0
Accept: text/html,application/xhtml+xml,application/xml;
        q=0.9,*/*;q=0.8
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Portanto, os seguintes cabeçalhos são enviados: `Host` , `User-Agent` , `Accept` , `Accept-Language` , `Accept-Encoding` e `Connection` . Cada um desses tem um significado para o servidor – no entanto, o protocolo HTTP não exige nenhum deles. Ao estabelecer esta liberdade, tanto o cliente quanto o servidor estão livres para negociar o conteúdo da maneira como acharem melhor.

Obviamente, isto não quer dizer que estes cabeçalhos não são padronizados. Da lista anterior, cada cabeçalho possui uma

explicação:

- `Host` – mostra qual foi o DNS utilizado para chegar a este servidor;
- `User-Agent` – fornece informações sobre o meio utilizado para acessar este endereço;
- `Accept` – realiza negociação com o servidor a respeito do conteúdo aceito (mais informações na seção *Media types*);
- `Accept-Language` – negocia com o servidor qual o idioma a ser usado na resposta;
- `Accept-Encoding` – negocia com o servidor qual a codificação a ser utilizada na resposta;
- `Connection` – ajusta o tipo de conexão com o servidor (persistente ou não).

Tecnicamente falando, os cabeçalhos são usados para tráfego de *metadados* – ou seja, informações a respeito da informação "de verdade". Tome um cenário de envio de e-mails: o destinatário, o assunto e outras informações são apenas *metadados* – a verdadeira informação é o seu conteúdo. Desta forma, uma requisição para um serviço HTTP de envio de e-mails poderia ser feita da seguinte forma:

```
POST /email HTTP/1.1
Host: geladaonline.com.br
Destinatario: aleaudate@gmail.com
Assunto: Quanto custa a cerveja dos Simpsons?
```

Vários usos diferentes são dados para cabeçalhos em HTTP. Ao longo deste livro, vou mencioná-los quando necessário.

2.7 MEDIA TYPES

Ao realizar uma requisição para o site <http://www.casadocodigo.com.br>, o Media Type `text/html` é

usado, indicando para o navegador qual é o tipo da informação que está sendo trafegada (no caso, HTML). Isto é utilizado para que o cliente saiba como trabalhar com o resultado (e não com tentativa e erro, por exemplo). Dessa maneira, os Media Types são formas padronizadas de descrever uma determinada informação (BORENSTEIN; FREED, 1996).

Os Media Types são divididos em *tipos* e *subtipos*, e acrescidos de parâmetros (se houver). São compostos com o seguinte formato: **tipo/subtipo**. Se houver parâmetros, o ; (ponto e vírgula) será utilizado para delimitar a área deles.

Portanto, um exemplo de Media Type seria `text/xml; charset="utf-8"` (para descrever um XML cuja codificação seja UTF-8).

Cada um desses tipos é utilizado com diferentes propósitos. Os mais comuns são:

- `application` – usado para tráfego de dados específicos de certas aplicações.
- `audio` – usado para formatos de áudio.
- `image` – usado para formatos de imagens.
- `text` – usado para formatos de texto padronizados ou facilmente inteligíveis por humanos.
- `video` – usado para formatos de vídeo.
- `vnd` – para tráfego de informações de softwares específicos (por exemplo, o Microsoft Office).

Em serviços REST, vários tipos diferentes de Media Types são utilizados. As maneiras mais comuns de representar dados estruturados, em serviços REST, são via XML e JSON, que são representados pelos Media Types `application/xml` e `application/json`, respectivamente. O XML também pode ser representado por `text/xml`, desde que possa ser considerado

legível por humanos – de acordo com a RFC 3023 (KOHN; MURATA; LAURENT, 2001).

Note que subtipos mais complexos do que isso podem ser representados com o sinal + (mais). Este sinal é utilizado em vários subtipos para delimitação de mais de um subtipo. Este é o caso com XHTML, por exemplo, que denota o tipo HTML acrescido das regras de XML, e cujo media type é application/xhtml+xml . Outro caso comum é o do protocolo SOAP, cujo media type é application/soap+xml .

Os Media Types são negociados a partir dos cabeçalhos Accept e Content-Type . O primeiro é utilizado em requisições, e o segundo, em respostas.

Ao usar o cabeçalho Accept , o cliente informa ao servidor qual tipo de dados espera receber. Caso seja o tipo de dados que não possa ser fornecido pelo servidor, este retorna o código de erro 415 (apresentado na seção 415 – *Unsupported Media Type*), indicando que o *Media Type* não é suportado. Se o tipo de dados existir, então os dados são fornecidos e o cabeçalho Content-Type apresenta qual é o tipo de informação fornecida.

Existem várias formas de realizar essa solicitação. Caso o cliente esteja disposto a receber mais de um tipo de dados, o cabeçalho Accept pode ser usado para esta negociação. Por exemplo, se o cliente puder receber qualquer tipo de imagem, esta solicitação pode utilizar um **curinga**, representado por * (asterisco). O servidor converterá essa solicitação em um tipo adequado. Essa negociação será similar à seguinte:

- Requisição:

```
GET /foto/1 HTTP/1.1
Host: geladaonline.com.br
Accept: image/*
```

- Resposta:

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 10248
```

Além disso, também é possível solicitar mais de um tipo de dados apenas separando-os por , (vírgula). Por exemplo, um cliente de uma página web poderia realizar uma solicitação como a seguinte:

```
GET / HTTP/1.1
Host: geladaonline.com.br
Accept:text/html,*/*
```

Note, assim, que o cliente solicita uma página HTML, se uma não estiver disponível, qualquer tipo de conteúdo é aceito – como pode ser observado pela presença de dois curingas, em tipo e subtipo de dados. Neste caso, a prioridade atendida é a da **especificidade**, ou seja, como `text/html` é mais específico, tem mais prioridade.

Em caso de vários tipos de dados que não atendam a esta regra, o parâmetro `q` pode ser adotado para que se possam realizar diferenciações de peso. Ele recebe valores variando entre 0.1 e 1, contendo a ordem de prioridade. Por exemplo, suponha a seguinte requisição:

```
GET / HTTP/1.1
Host: geladaonline.com.br
Accept:text/html,application/xhtml+xml,application/xml;
        q=0.9,*/*;q=0.8
```

Isto significa que tanto HTML quanto XHTML têm prioridade máxima. Na falta de uma representação do recurso com qualquer um desses tipos de dados, o XML é aceito com prioridade 90%. Na ausência do XML, qualquer outra representação é aceita, com prioridade 80%.

2.8 CÓDIGOS DE STATUS

Toda requisição que é enviada para o servidor retorna um código de status. Esses códigos são divididos em cinco famílias: 1xx, 2xx, 3xx, 4xx e 5xx, sendo:

- 1xx – Informacionais;
- 2xx – Códigos de sucesso;
- 3xx – Códigos de redirecionamento;
- 4xx – Erros causados pelo cliente;
- 5xx – Erros originados no servidor.

Alguns dos códigos mais importantes e mais utilizados são:

2xx

200 – OK

Indica que a operação indicada teve sucesso.

201 – Created

Indica que o recurso desejado foi criado com sucesso. Deve retornar um cabeçalho `Location`, que deve conter a URL onde o recurso recém-criado está disponível.

202 – Accepted

Indica que a solicitação foi recebida e será processada em outro momento. É tipicamente utilizada em requisições assíncronas, que não serão processadas em tempo real. Por esse motivo, pode retornar um cabeçalho `Location`, que trará uma URL na qual o cliente pode consultar se o recurso já está disponível ou não.

204 – No Content

Usualmente enviado em resposta a uma requisição `PUT` , `POST` ou `DELETE` , onde o servidor pode se recusar a enviar conteúdo.

3xx

301 – Moved Permanently

Significa que o recurso solicitado foi realocado permanentemente. Uma resposta com o código 301 deve conter um cabeçalho `Location` com a URL completa (ou seja, com descrição de protocolo e servidor) de onde o recurso está atualmente.

303 – See Other

É utilizado quando a requisição foi processada, mas o servidor não deseja enviar o resultado do processamento. Em vez disso, o servidor envia a resposta com este código de status e o cabeçalho `Location` , informando onde a resposta do processamento está.

304 – Not Modified

É usado, principalmente, em requisições `GET` condicionais – quando o cliente deseja ver a resposta apenas se ela tiver sido alterada em relação a uma requisição anterior. Veremos mais a respeito no capítulo *Tópicos avançados de serviços REST*.

307 – Temporary Redirect

Similar ao 301, mas indica que o redirecionamento é temporário, não permanente.

4xx

400 – Bad Request

É uma resposta genérica para qualquer tipo de erro de processamento cuja responsabilidade é do cliente do serviço.

401 – Unauthorized

Utilizado quando o cliente está tentando realizar uma operação sem ter fornecido dados de autenticação (ou a autenticação fornecida for inválida). Veremos mais a respeito no capítulo *Segurança REST*.

403 – Forbidden

Usado quando o cliente está tentando realizar uma operação sem ter a devida autorização.

404 – Not Found

Utilizado quando o recurso solicitado não existe.

405 – Method Not Allowed

Usado quando o método HTTP utilizado não é suportado pela URL. Deve incluir um cabeçalho `Allow` na resposta, contendo a listagem dos métodos suportados (separados por vírgula).

409 – Conflict

Utilizado quando há conflitos entre dois recursos. Comumente usado em resposta a criações de conteúdos que tenham restrições de dados únicos – por exemplo, criação de um usuário no sistema utilizando um *login* já existente. Se for causado pela existência de outro recurso (como no caso citado), a resposta deve conter um cabeçalho `Location`, explicitando a localização do recurso que é a fonte do conflito.

410 – Gone

Semelhante ao 404, mas indica que um recurso já existiu neste local.

415 – Unsupported Media Type

Usado em resposta a clientes que solicitam um tipo de dados que não é suportado – por exemplo, solicitar JSON quando o único formato de dados suportado é XML.

5xx

500 – Internal Server Error

É uma resposta de erro genérica, utilizada quando nenhuma outra se aplica.

503 – Service Unavailable

Indica que o servidor está atendendo requisições, mas o serviço em questão não está funcionando corretamente. Pode incluir um cabeçalho `Retry-After`, dizendo ao cliente quando ele deveria tentar submeter a requisição novamente.

2.9 CONCLUSÃO

Você pôde conferir, neste capítulo, os princípios básicos do protocolo HTTP. Estes são o uso de cabeçalhos, métodos, códigos de status e formatos de dados – que formam a base do uso eficiente de REST.

Você vai conferir, nos próximos capítulos, como REST se beneficia dos princípios deste protocolo de forma a tirar o máximo proveito desta técnica.

CAPÍTULO 3

CONCEITOS DE REST

"A mente que se abre a uma nova ideia jamais volta a seu tamanho original." – Albert Einstein

Como dito anteriormente, REST é baseado nos conceitos do protocolo HTTP. Além disso, este protocolo é a base para a *web* como a conhecemos, sendo que a própria navegação nela pode ser encarada como um uso de REST.

No entanto, REST não é tão simples quanto simplesmente utilizar HTTP. Existem regras que devem ser seguidas para se realizar uso efetivo deste protocolo. A intenção deste capítulo é apresentar quais são estes conceitos, de maneira que você, leitor, conheça as técnicas certas para desenvolvimento deste tipo de serviço.

3.1 SEMÂNTICAS DE RECURSOS

Todo serviço REST é baseado nos chamados **recursos**, que são entidades bem definidas em sistemas, que possuem identificadores e endereços (URLs) próprios. No caso da aplicação que estamos desenvolvendo, `geladaonline.com.br` , podemos assumir que uma cerveja é um recurso. Assim, as regras de REST dizem que as cervejas devem ter uma URL própria e que esta URL deve ser significativa, ou seja, descrever objetivamente o recurso. Desta forma, uma boa URL para cervejas pode ser `/cervejas` .

URLS NO SINGULAR OU NO PLURAL?

Não existem regras em relação a estas URLs estarem no singular ou no plural – ou seja, não importa se você prefere usar `/cerveja` ou `/cervejas`. O ideal, no entanto, é manter um padrão: tenha todas as suas URLs no singular ou todas no plural, mas não misture.

De acordo com o modelo REST, esta URL realizará interação com todas as cervejas do sistema. Para tratar de cervejas específicas, são usados identificadores. Estes podem ter qualquer formato – por exemplo, suponha uma cerveja Erdinger Weissbier com código de barras 123456. Note que o nome da cerveja não serve como identificador (existem várias "instâncias" de Erdinger Weissbier), mas o código de barras, sim.

Desta forma, deve ser possível buscar esta cerveja pelo código de barras, com a URL `/cervejas/123456`. Lembre-se, regras semelhantes aplicam-se a **qualquer** recurso. O identificador a ser utilizado na URL pode ser qualquer coisa que você assim desejar, e mais de um tipo de identificador pode ser usado para alcançar um recurso.

Por exemplo, suponha que a sua base de clientes tenha tanto um identificador gerado pelo banco de dados quanto os CPFs dos clientes:

- **Cliente número 1:** CPF 445.973.986-00
- **Cliente número 2:** CPF 428.193.616-50
- **Cliente número 3:** CPF 719.760.617-92

Assim, para buscar o cliente de número 1, deve ser possível usar

tanto a URL `/cliente/1` quanto a URL `/cliente/445.973.986-00`.

Pode haver casos, também, de identificadores compostos. Neste caso, a melhor prática seria separá-los utilizando ; (ponto e vírgula). Por exemplo, um serviço de mapas em que seja possível achar um ponto por latitude e longitude teria URLs como `/local/-23.5882888;-46.6323259`.

No capítulo *Tópicos avançados de serviços REST*, detalharei casos mais complexos de URLs.

3.2 INTERAÇÃO POR MÉTODOS

Para interagir com as URLs, os métodos HTTP são utilizados. A regra de ouro para esta interação é que **URLs são substantivos, e métodos HTTP são verbos**. Isto quer dizer que os métodos HTTP são os responsáveis por provocar alterações nos recursos identificados pelas URLs.

Estas modificações são padronizadas, de maneira que:

- GET – recupera os dados identificados pela URL;
- POST – cria um novo recurso;
- PUT – atualiza um recurso;
- DELETE – apaga um recurso.

CRUD – Create, Retrieve, Update, Delete

Note que esses quatro métodos principais podem ser diretamente relacionados a operações de bancos de dados. Assim, para recuperar o cliente de número 1 do banco de dados, basta usar o método GET em conjunto com a URL `/cliente/1`. Para criar um novo cliente, basta utilizar o método POST sobre a URL `/cliente` (o identificador será criado pelo banco de dados). Já

para atualizar este cliente, use o método `PUT` sobre a URL `/cliente/1` e, finalmente, para apagar o cliente, utilize o método `DELETE` sobre a URL `/cliente/1`.

Note que todas estas operações são **lógicas**. Isto quer dizer que utilizar o método `DELETE`, por exemplo, não significa necessariamente excluir o dado do banco de dados – significa apenas indisponibilizar o recurso para consumo pelo método `GET`. Ou seja, `DELETE` pode apenas marcar o dado no banco de dados como **desativado**.

Execução de funções

Criar serviços REST que executem tarefas de negócio (enviar um e-mail, criar uma máquina em um serviço de *cloud computing*, validar um CPF e outras tarefas que não necessariamente envolvam interação com um banco de dados) é uma tarefa mais complexa. Exige certo grau de domínio da teoria sobre REST, e muitas vezes não existem respostas 100% assertivas em relação à corretude da solução.

Para obter resultados satisfatórios, deve-se sempre ter o pensamento voltado para a orientação a recursos. Tomemos como exemplo o caso do envio de e-mails: a URL deve ser modelada em formato de substantivo, ou seja, apenas `/email`. Para enviar o e-mail, o cliente pode usar o método `POST` – ou seja, como se estivesse "criando" um e-mail.

Da mesma forma, inicializar uma máquina em um serviço de *cloud computing* pode ter uma abordagem mais similar a um CRUD: para inicializar a máquina, o cliente utiliza o método `POST` na URL `/maquinas`. Para desligá-la, utiliza o método `DELETE`.

Existem também casos em que as soluções podem não parecer triviais à primeira vista. Um caso clássico é o de validações (de CPF,

por exemplo). A URL pode ser claramente orientada a substantivos (`/validacao/CPF`, por exemplo), mas e quanto aos métodos?

A solução para este caso é pensar da seguinte maneira: *eu preciso obter uma validação? Ou criar uma, ou apagar, ou atualizar?*. Desta maneira, a resposta natural para a questão parece convergir para o método `GET`. Note que, neste caso, entra em cena o uso do método `HEAD`. Poderíamos usar o método `GET` para conseguir resultados detalhados, e o método `HEAD` para obter apenas o código de *status* da validação (lembre-se de que este método não retorna corpo!).

3.3 REPRESENTAÇÕES DISTINTAS

Um dos pilares de REST é o uso de *media types* para alterar as representações de um mesmo conteúdo sob perspectivas distintas. Esta ótica fica evidente quando se responde à pergunta: *que lado do recurso que estou procurando eu quero enxergar? Preciso de uma foto de uma cerveja, ou da descrição dela?*. Por exemplo, ao realizar uma busca por uma cerveja no sistema, pode-se tanto desejar um XML com os dados da cerveja quanto um JSON quanto uma foto da cerveja:



Figura 3.1: Pode ser mais interessante obter uma foto da cerveja do que a descrição dela

Com REST, tudo isso pode ser feito utilizando-se a mesma URL, por exemplo, `/cervejas/1`. O que vai modificar o resultado é a

solicitação que o cliente fizer por meio do cabeçalho `Accept` : se o cliente fizer uma solicitação para a URL `/cervejas/1` passando o cabeçalho `Accept` com o valor `application/xml` , ele vai obter uma representação da cerveja como XML. Mas se o cliente fizer esta requisição passando o valor `image/*` , ele obterá uma foto da cerveja.

O CABEÇALHO ACCEPT E OS CURINGAS

Note que, neste exemplo, estou falando de obter uma imagem passando o cabeçalho `Accept` com o valor `image/*` . Isto se dá devido ao fato de que, muitas vezes, não estamos interessados no tipo da imagem (JPG, GIF, PNG etc.), mas apenas no fato de ela **ser** uma imagem!

Leonard Richardson e Sam Ruby (2007) também defendem, em uma abordagem mais pragmática, que é interessante oferecer distinções entre esses tipos de dados na própria URL, a partir de extensões (similar às extensões de arquivos no sistema operacional). Por exemplo, no caso das cervejas, para obter uma representação das cervejas como XML, utilizaríamos a URL `/cervejas.xml` . Já para obter a representação como uma imagem `.jpg` , usariamos a URL `/cervejas.jpg` .

Note que essa abordagem possui vantagens e desvantagens. Ao passo que a testabilidade é simplificada (já que podemos testar os resultados pelo navegador), a implementação pode ser mais complexa e pode induzir diferenciações nas implementações (ou seja, ter lógicas distintas para buscar dados semelhantes).

3.4 USO CORRETO DE STATUS CODES

Algo que REST também prevê é o uso correto dos *status codes* HTTP. Na prática, isso significa conhecê-los e aplicá-los de acordo com a situação. Por exemplo, o código 200 (`OK`) é usado em grande parte das situações (o que pode acabar "viciando" o desenvolvedor), mas a criação de recursos deve retornar, quase sempre, o código 201 (`Created`) e o cabeçalho `Location`, indicando a localização do recurso criado:

The screenshot shows a browser window titled "Response". The URL is "POST on http://localhost:8080/cervejaria/cervejas". The status is "Status: 201 Created". The response body contains the following XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><cerveja>
<nome>Skol</nome><descricao>Cerveja brasileira</descricao>
<cervejaria>Ambev</cervejaria><tipo>PILSEN</tipo></cerveja>
```

Below the response body, there is a "Headers:" section with the following entries:

Content-Length	181
Location	http://localhost:8080/cervejaria/cervejas/Skol
Content-Type	application/xml
Server	Jetty(8.1.9.v20130131)

A "Close" button is located at the bottom right of the window.

Figura 3.2: Cabeçalho Location

Ainda, se dois clientes realizarem criação de recursos de forma que estes forneçam a chave utilizada para referenciar o recurso (ou seja, existe uma restrição de que a chave deve ser única por recurso), e estes dois clientes fornecerem a mesma chave na criação do recurso, o código de *status* 409 (*Conflict*) deve ser utilizado.

POSSO UTILIZAR A CHAVE PRIMÁRIA DO MEU BANCO DE DADOS COMO ID DOS RECURSOS?

É errado assumir que a chave usada nos recursos será **sempre** o ID do banco de dados. É perfeitamente aceitável utilizar esta chave, mas isto deve ser uma **coincidência**, de maneira que o seu cliente **jamais** se sinta como se estivesse simplesmente usando uma interface diferente para acessar um banco de dados.

Esta restrição é condizente com o que mencionei em meu livro *SOA Aplicado: integrando com web services e além* (2013), de que o cliente nunca deve sentir os efeitos de uma modificação na infraestrutura do serviço.

Por exemplo, suponha que a criação dos clientes na cervejaria utilize o CPF como chave dos recursos. Se dois clientes dos serviços tentarem se cadastrar com o mesmo CPF, um deles deve receber o código 409. A resposta com esse código deve conter o cabeçalho `Location` , indicando onde está o recurso que originou o conflito.

Ao longo do livro, apresentarei situações em que o uso correto dos códigos de *status* fará a diferença no consumo dos serviços.

3.5 HATEOAS

A última das técnicas mencionadas por Roy é o uso de *Hypermedia As The Engine Of Application State* – HATEOAS. Trata-se de algo que todo desenvolvedor web já conhece (apenas não por esse nome).

Toda vez que acessamos uma página web, além do texto da página, diversos links para outros recursos são carregados. Estes recursos incluem *scripts* JavaScript, CSS, imagens e outros. Além disso, muitas páginas têm formulários, que encaminham dados para outras URLs.

Por exemplo, considere o seguinte trecho de página HTML:

```
<html>
  <head>
    <link rel="icon" href="/assets/favicon.ico"
          type="image/ico" />
    <link href="/assets/fonts.css" rel="stylesheet"
          type="text/css" />
  </head>
  <body>
    
  </body>
</html>
```

Nela, existe uma referência para o ícone da página (referenciado com a tag `link` e atributo `rel="icon"`), uma referência para um arquivo CSS e uma referência para uma imagem.

HATEOAS considera estes links da mesma forma, por meio da referência a ações que podem ser tomadas a partir da entidade atual. Por exemplo, vamos modelar o sistema de compras da nossa cervejaria. Para criar um pedido de compra, é necessário fornecer um XML contendo os dados da requisição:

```
<compra>
  <item>
    <cerveja id="1">Stella Artois</cerveja>
    <quantidade>1</quantidade>
  </item>
</compra>
```

Se esta compra for aceita pelo servidor, o seguinte pode ser retornado:

```
<compra id="123">
  <item>
    <cerveja id="1">Stella Artois</cerveja>
    <quantidade>1</quantidade>
  </item>
  <link rel="pagamento" href="/pagamento/123" />
</compra>
```

Note a presença da tag `link`. Se o cliente utilizar a URL presente no atributo `href`, pode informar ao servidor o meio de pagamento aceito para a compra. A criação do meio de pagamento será feita pelo método HTTP `POST`, conforme mencionado anteriormente. Esse conhecimento "prévio" faz parte do que se convencionou chamar, em REST, de **interface uniforme**.

Estes links em HATEOAS têm dois segmentos: **links estruturais** ou **transicionais**. Os links estruturais, como diz o nome, são referentes à estrutura do próprio conteúdo. Por exemplo, suponha que você busque uma listagem de endereços de um determinado cliente:

```
<cliente>
  <enderecos>
    <link href="/cliente/1/endereco/1"
          title="Endereço comercial" />
    <link href="/cliente/1/endereco/2"
          title="Endereço residencial" />
    <link href="/cliente/1/endereco/3"
          title="Endereço alternativo" />
  </endereco>
</cliente>
```

Estes links produzem diversos benefícios para o cliente.

O primeiro benefício, e talvez o mais óbvio, é a redução do acoplamento entre o cliente e o servidor, já que o cliente pode seguir esses links para descobrir onde estão os endereços. Caso a URL de endereços mude, ele não deverá sentir os impactos se seguir os links.

O segundo benefício é que eles exercem uma ação inteligente do ponto de vista do cliente, já que este não precisa recuperar dados de que não precisa. Se ele precisar apenas do endereço comercial, basta seguir somente o link que contém o endereço comercial, e não todos os outros.

O terceiro é do ponto de vista da performance. Além de recuperar um volume menor de dados, em caso de uma lista (como neste exemplo, de endereços), o cliente pode paralelizar as requisições e, assim, diminuir o tempo de obtenção das informações. Obviamente, isso depende da latência da rede (ou seja, o tempo decorrido em relação à saída da requisição da máquina cliente e chegada ao servidor, e vice-versa), mas isso pode beneficiar-se do cacheamento do recurso (que veremos no capítulo *Tópicos avançados de serviços REST*).

Já os links transicionais são relativos a ações, ou seja, a ações que o cliente pode efetuar utilizando aquele recurso. Você já viu esse exemplo quando me referi ao pagamento da compra.

Existem cinco atributos importantes para os links:

- `href`
- `rel`
- `title`
- `method`
- `type`

EXISTE ALGUMA ESPECIFICAÇÃO PARA ESTES LINKS?

Existe uma especificação formal para tratamento destes links chamada XLink (disponível em <http://www.w3.org/TR/xlink/>). Os atributos `href`, `title` e `method` estão definidos lá. Os outros são usados como convenção.

O atributo `href` faz referência à URL onde o recurso está localizado. Apenas como lembrete, esta URL pode ser absoluta – com especificação de protocolo, *host*, porta etc. – ou relativa – ou seja, o cliente deve buscar o recurso no mesmo servidor onde fez a primeira requisição.

O atributo `rel` é usado com um texto de auxílio para o uso, que não deve ser lido pelo cliente final. O valor deste atributo deve ser utilizado pelo cliente para detectar o tipo de informação presente na URL. Por exemplo, o Netflix utiliza URLs para realizar esta distinção, algo como <http://schemas.netflix.com/catalog/people.directors>.

Caso tenha curiosidade, consulte a URL http://developer.netflix.com/docs/REST_API_Reference – onde está presente a documentação de referência para utilização da API do Netflix.

O atributo `title` contém uma descrição, legível por humanos, da informação que está presente na URL.

O atributo `method` é um dos tipos menos utilizados; quando o é, indica quais tipos de métodos HTTP são suportados pela URL (separados por vírgula).

Finalmente, o atributo `type` indica quais *media types* são

suportados pela URL, e também são dos menos usados.

3.6 BOAS PRÁTICAS DE PASSAGEM DE PARÂMETROS

A partir dos conceitos apresentados neste capítulo, muitas coisas devem estar mais claras a você, leitor. No entanto, como combinar estes conceitos para formar uma implementação correta?

Um dos pontos mais complexos quando nos referimos a uma implementação REST é, sem sombra de dúvidas, a questão de como passar parâmetros para nossos serviços. A seguir, enumero quais são estes e quando utilizá-los.

Path parameters

Os *path parameters* são parâmetros que são usados diretamente na URL (por exemplo, quando apresentei a URL `/cerveja/1`, o "1" foi oferecido como parâmetro). Estes devem ser utilizados quando o parâmetro para o serviço é **obrigatório**, isto é, quando a não passagem deste parâmetro provoca um erro no lado do servidor ou leva a um recurso diferente.

Por exemplo, se usarmos a URL `/cerveja`, listamos todas as cervejas do sistema; se utilizarmos a URL `/cerveja/1`, listamos apenas uma única cerveja – são, portanto, recursos diferentes.

Vale lembrar de que as URLs REST obedecem a uma hierarquia. Por exemplo, se quisermos obter informações a respeito da cervejaria de uma determinada cerveja, podemos ter a seguinte URL: `/cerveja/1/cervejaria`. Ou seja, a cervejaria é referente à cerveja de ID 1.

Query parameters

Os *query parameters* são parâmetros que também são utilizados diretamente na URL, mas que são apresentados após um ponto de interrogação (?) e delimitados por um "e" comercial (&), conforme visto em *Tipos de passagem de parâmetros*. São melhores usados em parâmetros que são opcionais, ou seja, que apenas provocam alterações na exibição do recurso e em caráter temporário.

Por exemplo, suponha que o nosso serviço de cervejas suporta paginação. Assim sendo, um *query parameter* seria uma boa pedida para fornecer o número da página, assim: `/cerveja?pagina=1` . Desta forma, o parâmetro é opcional (se não for fornecido, assume-se a primeira página) e provoca uma alteração apenas na exibição (ou seja, ao fornecer o número da página, quer dizer que apenas uma quantidade limitada de cervejas é fornecida no retorno).

Matrix parameters

Os *matrix parameters* são oferecidos de forma semelhante aos *query parameters*, exceto que são delimitados por ponto e vírgula (;). São utilizados quando o serviço recebe parâmetros relacionados entre si.

Por exemplo, suponha um serviço de mapas, que requeira o fornecimento de latitude e longitude, assim: `/cerveja;lat=0.414544;long=0.4556564` . Note que, caso esses parâmetros fossem fornecidos como *path parameters*, eles teriam de obedecer a uma ordem hierárquica, o que não é o caso aqui.

Form parameters

Os *form parameters* atendem à necessidade de se consumir serviços REST a partir de formulários HTML. Os parâmetros são inseridos no corpo da requisição e são enviados para o servidor com o mesmo formato que os *query parameters*. Eles devem também ser passados com o `type application/x-www-form-urlencoded` .

Head parameters

Os *head parameters* são passados para o servidor como cabeçalhos HTTP. Assim como todos os outros cabeçalhos, devem ser usados na passagem de metadados para o servidor. Estes metadados podem ser, por exemplo, o número da versão de uma determinada API:

```
GET /dados HTTP/1.1
Version: 1.0
```

Body parameters

Os *body parameters* (chamados também de **entidade** HTTP) são utilizados na passagem de dados em métodos específicos, como o POST. É no corpo da requisição que se encontram os dados nos quais estamos interessados, como uma imagem ou documento XML.

3.7 CONCLUSÃO

Você conheceu, neste capítulo, as técnicas básicas de REST – ou seja, o que é necessário conhecer antes de começar a trabalhar com REST. Obviamente, muitas questões ainda não foram respondidas: como lidar com clientes concorrentes? Como trabalhar com *cache*? Como modelar casos mais avançados?

Ao longo dos próximos capítulos, você conhecerá a resposta para estas e outras questões. Vamos em frente?

CAPÍTULO 4

TIPOS DE DADOS

"Conhecimento é poder." – Francis Bacon

Para trabalhar de maneira eficiente com seus dados, é necessário conhecer, antes, os tipos de dados mais comuns a serem utilizados — suas vantagens, desvantagens, usos mais comuns etc. Além disso, é preciso saber como usá-los na sua linguagem de programação.

4.1 APRENDA A LIDAR COM XML

XML é uma sigla que significa *eXtensible Markup Language*, ou **linguagem de marcação extensível**. Por ter esta natureza extensível, conseguimos expressar grande parte de nossas informações utilizando este formato.

XML é uma linguagem bastante semelhante a HTML (*HyperText Markup Language*), porém, com suas próprias particularidades. Por exemplo, todo arquivo XML tem **um e apenas um** elemento raiz (assim como HTML), com a diferença de que este elemento raiz é flexível o bastante para ter qualquer nome.

Por exemplo, se quisermos representar várias cervejas, precisamos encapsulá-las em um único elemento raiz (no exemplo a seguir, `cervejas`):

```
<cervejas>
  <cerveja>
    <nome>Stella Artois</nome>
  </cerveja>
```

```
<cerveja>
  <nome>Erdinger Weissbier</nome>
</cerveja>
</cervejas>
```

Além disso, um XML tem seções específicas para fornecimento de **instruções de processamento** — ou seja, seções que serão interpretadas por processadores de XML que, no entanto, não fazem parte dos dados. Estas seções recebem os nomes de **prólogo** (quando estão localizadas antes dos dados) e **epílogo**, quando estão localizadas depois. Por exemplo, é comum encontrar um prólogo que determina a versão do XML e o *charset* utilizado. Este prólogo tem o seguinte formato:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Ou seja, diferente de tags regulares de XML (que têm o formato `<tag></tag>`), uma instrução de processamento tem o formato `<?nome-da-instrução ?>`. No exemplo, estou especificando a versão de XML utilizada (1.0) e o *charset* (UTF-8).

Afora esta informação, as estruturas mais básicas em um XML são as tags e os **atributos**, de forma que um XML simples tem o seguinte formato:

```
<?xml version="1.0" encoding="UTF-8" ?>
<tag atributo="valor">conteúdo da tag</tag>
```

Como observado no exemplo, os dados transportados pela sua aplicação podem estar presentes tanto na forma de atributos como de conteúdo das tags. Em um exemplo mais próximo do *case* de cervejaria, uma cerveja pode ser transmitida da seguinte forma:

```
<?xml version="1.0" encoding="UTF-8" ?>
<cerveja id="1">
  <nome>Stella Artois</nome>
</cerveja>
```

4.2 FERRAMENTAL XML: CONHECENDO OS

XML SCHEMAS

Por ser um formato amplamente utilizado por diversos tipos de aplicação, XML contém diversos tipos de utilitários para vários fins, a saber:

- Validação de formato e conteúdo;
- Busca de dados;
- Transformação.

A ferramenta XML mais usada quando se trata de XML são os XML Schemas . Tratam-se de arquivos capazes de descrever o formato que um determinado XML deve ter (lembre-se, XML é flexível a ponto de permitir qualquer informação). Um XML Schema , como um todo, é análogo à definição de classes em Java: definem-se os pacotes (que, em um XML Schema , são definidos como *namespaces*) e as classes propriamente ditas (que, em XML Schemas , são os tipos).

Utiliza-se XML Schemas para que tanto o cliente quanto o servidor tenham um "acordo" a respeito do que enviar/receber (em termos da estrutura da informação).

Por exemplo, suponha que cada uma das cervejarias possua um ano de fundação, assim:

```
<cervejaria>
  <fundacao>1850</fundacao>
</cervejaria>
```

Se o cliente não tiver uma referência a respeito do que enviar, ele pode enviar os dados assim:

```
<cervejaria>
  <anoFundacao>1850</anoFundacao>
</cervejaria>
```

Note que, desta forma, a informação não seria inteligível do

ponto de vista do cliente e/ou do serviço. Assim, para manter ambos cientes do formato a ser utilizado, pode-se usar um XML Schema .

Um XML Schema simples pode ser definido da seguinte forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://geladaonline.com.br/comum/v1"
    xmlns:tns="http://geladaonline.com.br/comum/v1">
</schema>
```

Um XML Schema é sempre definido dentro da tag schema . Esta tag deve conter, obrigatoriamente, a referência para o XML Schema <http://www.w3.org/2001/XMLSchema> e o atributo targetNamespace , que aponta qual deve ser o namespace usado pelo XML que estiver sendo validado por este XML Schema . Além disso, por convenção, o próprio namespace é referenciado no documento pela declaração xmlns:tns , indicando que o prefixo tns poderá ser utilizado no escopo deste XML Schema . Esta prática não é obrigatória, mas é sempre recomendada.

A estrutura dos dados em XML Schemas são definidas em termos de **elementos**. Os elementos são usados para definir informações a respeito das tags, como: nome da tag, número de repetições permitido, quais subtags são permitidas, quais atributos uma tag deve ter etc.

Para definir um elemento dentro de um XML Schema , basta utilizar a tag element . Por exemplo, para definir uma tag nome num XML Schema , basta usar o seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://geladaonline.com.br/comum/v1"
    xmlns:tns="http://geladaonline.com.br/comum/v1">
    <element name="nome" type="string" />
</schema>
```

Conforme mencionado anteriormente, a definição dos dados

propriamente ditos é feita por meio de tipos. Estes são divididos entre simples e complexos, sendo os simples aqueles que são *strings*, datas, números ou derivados destes; já os complexos são definidos a partir da junção de elementos de tipos simples e/ou outros tipos complexos.

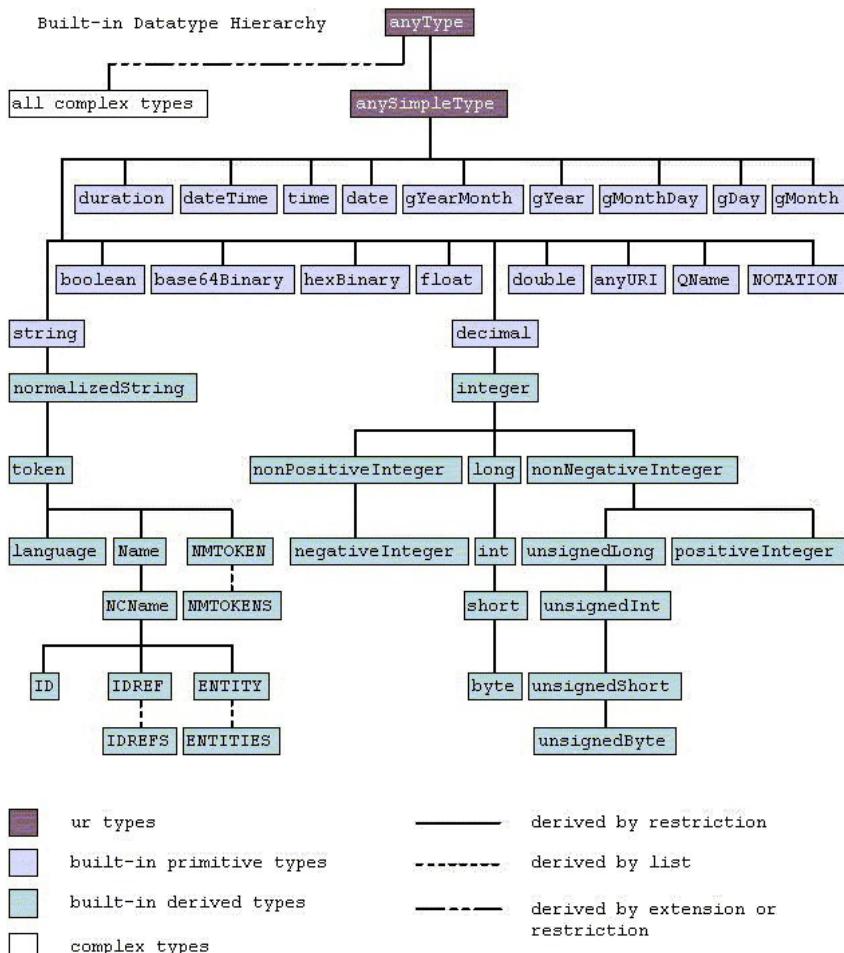


Figura 4.1: Descrição dos elementos embutidos em XML. Fonte:
<http://www.w3.org/TR/xmlschema-2/>

Tipos novos podem ser criados ou estendidos à vontade, sejam eles simples ou complexos. Por exemplo, para criar um tipo simples

novo, basta utilizar o seguinte:

```
<simpleType name="CEP" />
```

Neste mesmo schema, um tipo complexo pode ser criado da seguinte forma:

```
<complexType name="Endereco">
  <sequence>
    <element name="cep" type="tns:CEP" />
    <element name="logradouro" type="string" />
  </sequence>
</complexType>
```

Note que a definição do tipo complexo envolve a tag `sequence`. Esta tag é usada para determinar que os elementos nela envolvidos devem ser inseridos nesta ordem. Por exemplo, ao implementar este tipo complexo, o seguinte é aceito:

```
<endereco>
  <cep>12345-678</cep>
  <logradouro>Rua das cervejas</logradouro>
</endereco>
```

Porém, o inverso não é aceito:

```
<endereco>
  <logradouro>Rua das cervejas</logradouro>
  <cep>12345-678</cep>
</endereco>
```

Definindo formas de validação para tipos simples

Os tipos simples, como dito antes, oferecem extensibilidade em relação a outros tipos simples. Esta extensão pode ser feita para os fins mais diversos, sendo que o uso mais comum desta facilidade é para prover restrições sobre estes tipos simples. Por exemplo, no caso do CEP, sabemos que este segue uma estrutura bem clara de validação, que é o padrão cinco dígitos (traço) três dígitos. Ou seja, uma expressão regular pode ser utilizada para esta validação.

Esta restrição é representada usando-se a tag `restriction`. Esta, por sua vez, possui um atributo `base`, que é utilizado para indicar qual será o "tipo pai" a ser usado por este dado. Por exemplo, no caso de um CEP, o tipo a ser utilizado como tipo pai será `string`. Assim, o elemento `CEP` pode ser representado da seguinte forma:

```
<simpleType name="CEP">
    <restriction base="string">
        </restriction>
</simpleType>
```

Finalmente, dentro da tag `restriction` é que podemos definir o tipo de restrição que será aplicada ao utilizar este tipo. Podemos definir uma série de restrições, como enumerações de dados (ou seja, só é possível usar os valores especificados), comprimento máximo da string, valores máximos e mínimos que números podem ter etc.

Como queremos utilizar uma expressão regular, usamos a tag `pattern`, que define um atributo `value`. Por meio deste atributo, especificamos a expressão regular.

Assim sendo, definimos o nosso tipo CEP da seguinte forma:

```
<simpleType name="CEP">
    <restriction base="string">
        <pattern value="\d{5}-\d{3}" />
    </restriction>
</simpleType>
```

EXPRESSÕES REGULARES

Expressões regulares são usadas para determinar formatos que certas strings devem ter. Por exemplo, estas expressões podem ser utilizadas para validar endereços de e-mail, números de cartão de crédito, datas etc. Por ser um assunto demasiado comprido (que rende livros apenas sobre isso), limito-me aqui a apenas explicar o significado desse padrão.

O símbolo `\d` significa um dígito (qualquer um). Ao ter o sinal `{5}` anexado, indica que cinco dígitos são aceitos. O traço representa a si próprio.

Explorando tipos complexos

Como apresentado anteriormente, um tipo complexo em um XML Schema é semelhante a uma classe Java. Por exemplo, a definição de um endereço pode ser feita da seguinte forma:

```
<complexType name="Endereco">
    <sequence>
        <element name="CEP" type="tns:CEP" />
        <element name="logradouro" type="string" />
    </sequence>
</complexType>
```

Note a referência ao tipo `"CEP"`, definido anteriormente. O prefixo `tns`, conforme mencionado, faz referência ao namespace do próprio arquivo (semelhante a uma referência a um pacote Java — que, no entanto, possui uma forma abreviada, que é o prefixo).

Os tipos complexos comportam diversas facilidades, como herança e definição de atributos. Por exemplo, considere o tipo complexo `Pessoa`:

```
<complexType name="Pessoa">  
</complexType>
```

Supondo que uma Pessoa seja uma superdefinição para uma pessoa física ou jurídica, queremos que o tipo Pessoa seja abstrato. Assim, podemos definir o atributo abstract :

```
<complexType name="Pessoa" abstract="true">  
</complexType>
```

Note que, desta forma, não haverá uma implementação do que for definido neste tipo, apenas de seus subtipos. Para criar uma extensão deste tipo, utilizam-se as tags complexContent e extension , assim:

```
<complexType name="PessoaFisica">  
    <complexContent>  
        <extension base="tns:Pessoa">  
  
            </extension>  
        </complexContent>  
</complexType>
```

É possível definir elementos tanto no supertipo quanto no subtipo. Por exemplo, considere o seguinte:

```
<complexType name="Pessoa" abstract="true">  
    <sequence>  
        <element name="nome" type="string" />  
    </sequence>  
</complexType>  
  
<complexType name="PessoaFisica">  
    <complexContent>  
        <extension base="tns:Pessoa">  
            <sequence>  
                <element name="cpf" type="tns:CPF" />  
            </sequence>  
        </extension>  
    </complexContent>  
</complexType>
```

A implementação deste tipo pode ficar assim:

```
<pessoaFisica>
```

```
<nome>Alexandre</nome>
<cpf>123.456.789-09</cpf>
</pessoaFisica>
```

Além disso, é possível definir atributos nos tipos complexos. Por exemplo, considere o seguinte:

```
<complexType name="Pessoa" abstract="true">
    <attribute name="id" type="long" />
</complexType>
```

A implementação deste tipo ficaria assim:

```
<pessoaFisica id="1" />
```

Um XML Schema também pode importar outros, notavelmente para realizar composições entre vários tipos diferentes. Por exemplo, considere as duas definições de XML Schemas :

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://geladaonline.com.br/endereco/v1"
    elementFormDefault="qualified"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://geladaonline.com.br/endereco/v1">

    <simpleType name="CEP">
        <restriction base="string">
            <pattern value="\d{5}-\d{3}" />
        </restriction>
    </simpleType>

    <complexType name="Endereco">
        <sequence>
            <element name="cep" type="tns:CEP" />
            <element name="logradouro" type="string" />
        </sequence>
    </complexType>
</schema>
```

Note que este XML Schema possui o targetNamespace definido como <http://geladaonline.com.br/endereco/v1>.

Agora, considere um segundo XML Schema :

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://geladaonline.com.br/pessoa/v1"
```

```

elementFormDefault="qualified"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://geladaonline.com.br/pessoa/v1">

<simpleType name="CPF">
    <restriction base="string">
        <pattern value="\d{3}.\d{3}.\d{3}-\d{2}" />
    </restriction>
</simpleType>

<complexType name="Pessoa" abstract="true">
    <sequence>
        <element name="nome" type="string" />
    </sequence>
    <attribute name="id" type="long" />
</complexType>

<complexType name="PessoaFisica">
    <complexContent>
        <extension base="tns:Pessoa">
            <sequence>
                <element name="cpf" type="tns:CPF" />
            </sequence>
        </extension>
    </complexContent>
</complexType>
</schema>

```

Note que, neste segundo XML Schema , o namespace é <http://geladaonline.com.br/pessoa/v1> (ou seja, diferente do namespace de endereços).

Para usar o XML Schema de endereços no XML Schema de pessoas, é necessário efetuar dois passos: o primeiro é definir um prefixo para o namespace de endereços. Por exemplo, para definir o prefixo `end` , utiliza-se o seguinte:

```

<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://geladaonline.com.br/pessoa/v1"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://geladaonline.com.br/pessoa/v1"
    xmlns:end="http://geladaonline.com.br/endereco/v1">

    <!-- restante -->
</schema>

```

O próximo passo é informar ao mecanismo a localização do outro XML Schema pela tag `import` :

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://geladaonline.com.br/pessoa/v1"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://geladaonline.com.br/pessoa/v1"
  xmlns:end="http://geladaonline.com.br/endereco/v1">

  <import namespace="http://geladaonline.com.br/endereco/v1"
    schemaLocation="Endereco.xsd" />

  <!-- restante -->
</schema>
```

Assim, para usar os tipos definidos no novo arquivo, basta utilizar o prefixo `end` :

```
<complexType name="Pessoa">
  <sequence>
    <element name="endereco" type="end:Endereco" />
  </sequence>
</complexType>
```

Os tipos complexos também podem definir o número de ocorrências de seus elementos. Por exemplo, para determinar que um elemento tem número mínimo de ocorrências zero (ou seja, é opcional), utiliza-se a tag `minOccurs` :

```
<complexType name="Pessoa">
  <sequence>
    <element name="endereco" type="end:Endereco"
      minOccurs="0"/>
  </sequence>
</complexType>
```

Da mesma forma, é possível usar a tag `maxOccurs` para determinar o número máximo de ocorrências (ou seja, que um dado elemento representa uma lista). O número máximo de ocorrências pode ser delimitado a partir de um número fixo ou, caso não haja um limite definido, utiliza-se o valor `unbounded`. Assim, para determinar que uma pessoa possui vários endereços, pode-se

utilizar o seguinte:

```
<complexType name="Pessoa">
    <sequence>
        <element name="endereco" type="end:Endereco"
            maxOccurs="unbounded"/>
    </sequence>
</complexType>
```

Isso provoca o seguinte resultado:

```
<pessoaFisica>
    <endereco>
        <cep>12345-678</cep>
        <logradouro>Rua Um</logradouro>
    </endereco>
    <endereco>
        <cep>87654-321</cep>
        <logradouro>Rua Dois</logradouro>
    </endereco>
</pessoaFisica>
```

A este ponto, você deve ter notado que o elemento raiz `pessoaFisica` não foi especificado. O elemento raiz é definido fora de tipos em um XML Schema , usando a tag `element` . Desta forma, é possível ter o seguinte:

```
<complexType name="PessoaFisica">
    <!-- definição de pessoa física -->
</complexType>

<element name="pessoaFisica" type="tns:PessoaFisica" />
```

4.3 TRABALHANDO COM XML UTILIZANDO JAXB

A API (*Application Programmer Interface*) padrão para trabalhar com XML, em Java, é o JAXB (*Java Architecture for XML Binding*). Esta API trabalha essencialmente com anotações sobre classes Java, que dão instruções a respeito de como converter os dados em XML por meio da geração de XML Schemas .

Para transformar um XML Schema em classes Java compatíveis com JAXB, existe um utilitário presente na JDK chamado xjc . Supondo que seus XML Schemas estejam estruturados com o Maven, ou seja, presentes em uma pasta `src/main/resources` , é possível colocar estas classes na pasta certa usando o argumento `-d` :

```
xjc -d ../java Pessoa.xsd
```

O que deve produzir uma estrutura como a seguinte:

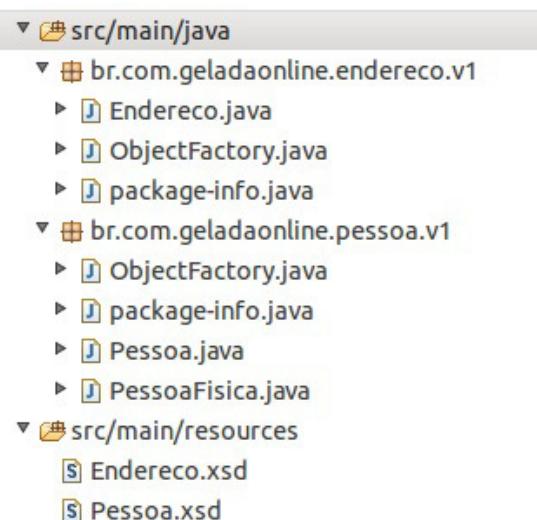


Figura 4.2: Arquivos gerados pelo xjc

Como você pode observar, uma classe para cada tipo complexo presente nos XML Schemas foi gerada, além das classes `ObjectFactory` e o arquivo `package-info.java` . Eles atuam, respectivamente, como uma classe fábrica para objetos recém-criados (ou seja, será utilizado no momento da tradução XML->Java) e como um mecanismo de fornecimento de informações válidas para todo o pacote.

O ARQUIVO PACKAGE-INFO.JAVA

Este arquivo é padronizado pela especificação Java (ou seja, não é definido pela especificação JAXB). Ele contém apenas a declaração de nome do pacote, com possíveis anotações. No caso do JAXB, este arquivo é gerado com a anotação `@XMLSchema`, que traz informações que serão válidas para todas as classes do pacote, como o namespace que será aplicado.

Vamos analisar as classes geradas:

A classe PessoaFisica

A classe `PessoaFisica` deve ter sido gerada com o seguinte código:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "PessoaFisica", propOrder = {
    "cpf"
})
public class PessoaFisica
    extends Pessoa
{
    @XmlElement(required = true)
    protected String cpf;

    // getter e setter para cpf
}
```

A anotação `@XMLAcessorType` indica para o JAXB qual o mecanismo a ser usado para realizar a tradução dos dados para XML. Como a engine do JAXB utiliza a API de *reflections* para realizar este mapeamento, o acesso aos dados pode ser feito tanto diretamente pelo valor dos atributos presentes na classe (sejam estes

privados ou não) como por *getters* e *setters*.

A API disponibiliza quatro formas distintas de acesso:

- **Por campos:** mapeia todos os atributos de classe, independente de sua visibilidade e de estarem anotados com anotações do JAXB. Caso não estejam mapeados com nenhuma informação a respeito de nomenclatura, serão mapeados com o nome do próprio campo. Ficam excluídos desta apenas atributos estáticos ou que sejam definidos como `transient`.
- **Por propriedades:** a engine detecta *getters* e *setters*, e mapeia todos os pares encontrados.
- **Por membros públicos:** mapeia todos os *getters* e *setters* que tenham visibilidade ajustada para `public` e também todos os atributos de classe públicos. Este é o padrão, quando nenhum dos outros tipos é definido.
- **Nenhum:** mapeia apenas os membros que estejam anotados com anotações do JAXB.

A anotação `XmlType` é usada no momento da conversão das classes em `XML Schemas`. Ela faz menção direta ao uso de tipos complexos em `XML Schemas`, e é utilizada, neste contexto, para descrever o nome do tipo complexo e a ordem das propriedades (que devem, obrigatoriamente, estar presentes na classe).

A anotação `XmlElement` é usada para declarar a propriedade como elemento (que é o comportamento padrão) e outras propriedades a respeito do elemento, como a obrigatoriedade de ter conteúdo, o nome do elemento, valor padrão etc. Neste caso, como o elemento `cpf` não teve declarado o número mínimo de ocorrências como zero, foi assumido o valor padrão um. Assim, o JAXB entende o elemento como obrigatório.

A classe Pessoa

A classe `Pessoa` deve ter o seguinte código:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Pessoa", propOrder = {
    "nome",
    "endereco"
})
@XmlSeeAlso({
    PessoaFisica.class
})
public abstract class Pessoa {

    @XmlElement(required = true)
    protected String nome;
    protected List<Endereco> endereco;
    @XmlAttribute(name = "id")
    protected Long id;

    // getters e setters

}
```

Como você pode notar, a classe `Pessoa` é abstrata. Isto é devido ao fato de o tipo complexo ter sido declarado também como abstrato, e as duas coisas são equivalentes em cada contexto (isto é, uma classe abstrata não poder ser instanciada em Java, e um tipo complexo não poder ser diretamente utilizado em XML).

Sendo uma classe abstrata, o JAXB precisa saber quais são suas subclasses para que a engine tenha condições de fazer o mapeamento XML->Java adequadamente. Como a própria linguagem Java não permite tal detecção, entra em cena a anotação `@XmlSeeAlso`, que indica para a engine JAXB quais classes estão relacionadas a esta (no caso, a classe `PessoaFisica`, vista anteriormente).

Outra anotação nova é `@XmlAttribute`. Esta anotação indica que o atributo da classe é mapeado como um atributo XML, e não como elemento (comportamento padrão).

Além disso, note a presença do atributo `endereco`, que foi definido como uma lista. Este comportamento é devido ao fato de que o elemento `endereco`, declarado com número mínimo de ocorrências igual a zero e sem limite superior — o que, automaticamente, caracteriza o elemento como uma lista.

Além disso, note que a classe `Endereco` está sendo utilizada, mas não declarada na anotação `@XmlSeeAlso`. Isto é devido ao fato de que, como esta classe já é referenciada pelo próprio código (ou seja, na declaração `List<Endereco>`), não é necessário realizar a declaração na anotação.

O arquivo package-info.java

O arquivo `package-info.java` deve ter sido gerado com o seguinte conteúdo:

```
@javax.xml.bind.annotation.XmlSchema
(namespace = "http://geladaonline.com.br/pessoa/v1",
elementFormDefault =
    javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
package br.com.geladaonline.pessoa.v1;
```

Note que, conforme explicado anteriormente, apenas a declaração do nome do pacote é permitida neste arquivo. Assim, a anotação `@XmlSchema` contém dados que são comuns a todas as classes do pacote, ou seja, a declaração do namespace e o formato dos dados, que é **qualificado**.

XML QUALIFICADO VERSUS XML NÃO QUALIFICADO

Quando dizemos que um XML é qualificado, nos referimos ao formato final com que ele será formado. Em um XML qualificado, todos os elementos devem mencionar a qual namespace pertencem, por exemplo:

```
<pes:pessoaFisica xmlns:pes=
    "http://geladaonline.com.br/pessoa/v1">
    <pes:cpf>123.456.789-09</pes:cpf>
</pes:pessoaFisica>
```

Note a presença do prefixo pes no atributo cpf . Se o XML não fosse qualificado, ele poderia ser escrito da seguinte maneira:

```
<pes:pessoaFisica xmlns:pes=
    "http://geladaonline.com.br/pessoa/v1">
    <cpf>123.456.789-09</cpf>
</pes:pessoaFisica>
```

Neste caso, o fato de não ser qualificado implica, automaticamente, que o namespace do elemento cpf é igual ao de pessoaFisica , ou seja, http://geladaonline.com.br/pessoa/v1 .

É sempre uma boa prática utilizar XMLs qualificados.

4.4 TESTANDO O XML GERADO

Para testar o formato dos dados que será produzido pelo JAXB, existe uma classe utilitária chamada javax.xml.bind.JAXB . Ela tem métodos para realizar *marshal* (ou seja, transformação das classes Java em XML) e *unmarshal* (ou seja, transformação de XML em Java).

ALERTA SOBRE A CLASSE JAXX.XML.BIND.JAXB

Como pode ser conferido na própria documentação desta classe, esta deve ser utilizada apenas para testes, não para produção.

Para realizar o teste, basta instanciar um objeto de uma classe anotada com JAXB e usar o método *marshal*, passando este objeto e uma *stream* como parâmetros. Por exemplo, para mostrar o resultado no console, basta utilizar o seguinte código:

```
public static void main(String[] args) {  
    PessoaFisica pessoaFisica = new PessoaFisica();  
    pessoaFisica.setCpf("12345678909");  
    pessoaFisica.setNome("Alexandre Saudate");  
  
    Endereco endereco = new Endereco();  
    endereco.setCep("12345-678");  
  
    pessoaFisica.getEndereco().add(endereco);  
  
    JAXB.marshal(pessoaFisica, System.out);  
}
```

De acordo com esse código, o seguinte deve ser apresentado:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>  
<pessoaFisica xmlns:ns2="http://geladaonline.com.br/endereco/v1"  
    xmlns:ns3="http://geladaonline.com.br/pessoa/v1">  
    <ns3:nome>Alexandre Saudate</ns3:nome>  
    <ns3:endereco>  
        <ns2:cep>12345-678</ns2:cep>  
    </ns3:endereco>  
    <ns3:cpf>12345678909</ns3:cpf>  
</pessoaFisica>
```

4.5 UTILIZANDO JAXB SEM UM XML SCHEMA

Muitas vezes, você pode considerar desnecessário usar um XML

Schema , ou muito trabalhoso gerar um. Nestes casos, é possível trabalhar com JAXB sem utilizar XML Schemas . Para isto, basta gerar as classes a serem usadas normalmente e utilizar a anotação javax.xml.bind.annotation.XmlRootElement .

Neste caso, podemos refatorar a classe PessoaFisica , por exemplo, para ser a raiz, e refatoramos toda a estrutura para remover os arquivos próprios do JAXB (ou seja, as classes ObjectFactory e package-info). A estrutura, então, fica no seguinte formato:



Figura 4.3: Nova estrutura

As classes também foram refatoradas, para ficar como a seguir:

```
@XmlRootElement
public class PessoaFisica
    extends Pessoa
{
    private String cpf;
    // getters e setters
}
```

A definição de Pessoa :

```
@XmlSeeAlso({
    PessoaFisica.class
})
public abstract class Pessoa {
```

```
private String nome;
private List<Endereco> endereco;
private Long id;

@XmlAttribute(name = "id")
public Long getId() {
    return id;
}

// getters e setters

}
```

E a classe que modelará o endereço:

```
public class Endereco {

    private String cep;
    private String logradouro;

}
```

O resultado do teste fica semelhante ao anterior:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<pessoaFisica id="1">
    <endereco>
        <cep>12345-678</cep>
    </endereco>
    <nome>Alexandre Saudate</nome>
    <cpf>12345678909</cpf>
</pessoaFisica>
```

4.6 JSON

JSON é uma sigla para *JavaScript Object Notation*. É uma linguagem de marcação criada por Douglas Crockford (2006), descrita na RFC 4627, e serve como uma contrapartida a XML. Tem por principal motivação o tamanho reduzido em relação a XML, e acaba tendo uso mais propício em cenários onde largura de banda (ou seja, quantidade de dados que pode ser transmitida em um determinado intervalo de tempo) é um recurso crítico.

Atende ao seguinte modelo:

- Um objeto contém zero ou mais membros;
- Um membro contém zero ou mais pares e zero ou mais membros;
- Um par contém uma chave e um valor;
- Um membro também pode ser um *array*.

O formato dessa definição é o seguinte:

```
{"nome do objeto" : {  
    "nome do par" : "valor do par"  
}  
}
```

Por exemplo, a pessoa física pode ser definida da seguinte maneira:

```
{"pessoaFisica" : {  
    "nome" : "Alexandre",  
    "cpf" : "123.456.789-09"  
}  
}
```

Caso seja uma listagem, o formato é o seguinte:

```
{"nome do objeto" : [  
    {"nome do elemento" : "valor"},  
    {"nome do elemento" : "valor"}  
]  
}
```

Novamente, a pessoa física pode ser definida da seguinte maneira:

```
{"pessoaFisica" : {  
    "nome" : "Alexandre",  
    [  
        "endereco" : {  
            "cep" : "12345-678",  
            "logradouro" : "Rua Um"  
        }  
    ],  
    "cpf" : "123.456.789-09"
```

```
    }  
}
```

Note que as quebras de linha e os espaços não são obrigatórios — servem apenas como elementos para facilitar a visualização.

Também vale a pena destacar que a declaração do elemento raiz (como a declaração `pessoaFisica`) é estritamente opcional. Muitas engines são capazes de trabalhar com apenas os atributos do elemento.

Por exemplo, a definição de pessoa física poderia ficar da seguinte forma:

```
{  
    "nome" : "Alexandre",  
    [  
        {  
            "cep" : "12345-678",  
            "logradouro" : "Rua Um"  
        }  
    ]  
    "cpf" : "123.456.789-09"  
}
```

4.7 TRABALHANDO COM JSON UTILIZANDO JAXB

Existem diversas implementações de *parsers* JSON para Java. Algumas delas são as seguintes:

- GSON
- Jackson
- Jettison
- XStream

A maneira de uso de cada uma tem vantagens e desvantagens. Uma vantagem bastante interessante de algumas é a capacidade de usar anotações JAXB como instruções para geração do JSON,

gerando apenas um único esforço quando queremos trabalhar tanto com JSON quanto com XML.

Para demonstrar o uso destas ferramentas, adicionei mais um campo na classe `PessoaFisica`, chamado `dadoTransiente`. Este campo foi anotado com `@XmlTransient`, indicando que este dado não deve ser apresentado quando o *parser* utilizar as anotações do JAXB, desta forma:

```
@XmlElement
public class PessoaFisica extends Pessoa {

    private String cpf;

    private String dadoTransiente = "dadoTransiente";

    @XmlTransient
    public String getDadoTransiente() {
        return dadoTransiente;
    }

    //getter e setters restantes
}
```

Vejamos o código com GSON. Para transformar uma instância desta classe em JSON com GSON, o código a seguir é utilizado:

```
PessoaFisica pessoaFisica = Parser.criarPessoaFisicaTeste();
Gson gson = new Gson();
System.out.println(gson.toJson(pessoaFisica));
```

O que produz o seguinte resultado (formatado por mim para oferecer melhor legibilidade):

```
{"cpf":"123.456.789-09",
"dadoTransiente":"dadoTransiente",
"nome":"Alexandre",
"endereco":[{"cep":"12345-678","logradouro":"Rua Um"}],
"id":1}
```

Como observado, é fácil gerar o conteúdo JSON utilizando GSON; no entanto, a informação transiente está presente no JSON

gerado, ou seja, o GSON não oferece suporte às anotações JAXB.

Já o código com Jackson fica da seguinte forma:

```
PessoaFisica pessoaFisica = Parser.criarPessoaFisicaTeste();
ObjectMapper objectMapper = new ObjectMapper();

AnnotationIntrospector annotationIntrospector =
    new JaxbAnnotationIntrospector();
objectMapper.setAnnotationIntrospector(annotationIntrospector);
System.out.println(objectMapper.
    writeValueAsString(pessoaFisica));
```

O que produz o seguinte resultado:

```
{"nome": "Alexandre",
"endereco": [{"cep": "12345-678", "logradouro": "Rua Um"}],
"id": 1,
"cpf": "123.456.789-09"
}
```

Desta forma, a informação transiente foi detectada e retirada do JSON. No entanto, o código está demasiado complicado.

Vamos criar um método para criar uma pessoa física de teste:

```
public static PessoaFisica criarPessoaFisicaTeste() {
    PessoaFisica pessoaFisica = new PessoaFisica();
    pessoaFisica.setCpf("123.456.789-09");
    pessoaFisica.setId(1L);
    pessoaFisica.setNome("Alexandre");

    Endereco endereco = new Endereco();
    endereco.setCep("12345-678");
    endereco.setLogradouro("Rua Um");

    List<Endereco> enderecos = new ArrayList<>();
    enderecos.add(endereco);

    pessoaFisica.setEndereco(enderecos);
    return pessoaFisica;
}
```

Agora, vejamos o código necessário para avaliar esta informação com Jettison:

```

PessoaFisica pessoaFisica = Parser.criarPessoaFisicaTeste();
JAXBContext context =
    JAXBContext.newInstance(PessoaFisica.class);
MappedNamespaceConvention con =
    new MappedNamespaceConvention();
Writer writer = new OutputStreamWriter(System.out);
XMLStreamWriter xmlStreamWriter =
    new MappedXMLStreamWriter(con, writer);
Marshaller marshaller = context.createMarshaller();
marshaller.marshal(pessoaFisica, xmlStreamWriter);

```

O código utilizado pelo Jettison é ainda mais complicado. No entanto, ele possui uma afinidade maior com o JAXB (como observado pelo uso das classes `JAXBContext` e `Marshaller`, que são próprias da API do JAXB). Isto é refletido no resultado da execução deste código:

```

{"pessoaFisica":
  {"@id":"1",
   "endereco": {"cep": "12345-678", "logradouro": "Rua Um"},
   "nome": "Alexandre",
   "cpf": "123.456.789-09"
  }
}

```

Note que o JSON gerado é mais semelhante a um XML. Está presente um elemento raiz, `pessoaFisica`, e o atributo `id` foi destacado por meio do prefixo `@`, indicando que este é equivalente a um atributo XML.

Finalmente, vejamos o código usado pelo XStream:

```

PessoaFisica pessoaFisica = Parser.criarPessoaFisicaTeste();
XStream xStream = new XStream(new JettisonMappedXmlDriver());
System.out.println(xStream.toXML(pessoaFisica));

```

Aqui, note o seguinte: o *driver* usado pelo XStream para geração do JSON é o do Jettison, ou seja, o XStream reutiliza outro framework para geração do JSON. O resultado, no entanto, é radicalmente diferente:

```
{"br.com.geladaonline.modelo.pessoa.PessoaFisica":
  {"nome": "Alexandre",
```

```

    "enderecos": [
        {"br.com.geladaonline.modelo.pessoa.Endereco":
            {
                "cep":"12345-678", "logradouro":"Rua Um"
            }
        }
    ],
    "id":1,
    "cpf":"123.456.789-09",
    "dadoTransiente":"dadoTransiente"
}
}

```

Note que o XStream, por padrão, coloca os nomes das classes utilizadas na tradução e não respeita as anotações do JAXB. Toda esta informação pode ser controlada, mas com código e anotações próprias do XStream.

4.8 VALIDAÇÃO DE JSON COM JSON SCHEMA

Assim como em XML, JSON também possui um sistema de validação, conhecido como **JSON Schema**. Trata-se de um tipo de arquivo que, como **XML Schemas**, também possui um formato próprio para especificação de tipos JSON. Essa especificação está disponível no seu site, <http://json-schema.org/>.

Um **JSON Schema** possui como declaração mais elementar o **título** (descrito no **JSON Schema** como uma propriedade `title`). O título é o que denomina este arquivo de validação. Por exemplo, supondo que desejamos começar a descrever um **JSON Schema** para pessoas, o descriptivo terá o seguinte formato:

```
{
    "title": "Pessoa"
}
```

Na sequência, é necessário definir para este formato o **tipo** (descrito no **JSON Schema** como uma propriedade `type`). O tipo pode ser um dos seguintes:

- `array` — ou seja, uma lista de dados;
- `boolean` — ou seja, um valor booleano (`true` ou `false`);
- `integer` — um número inteiro qualquer. Note que JSON não define um número de *bits* possível, apenas o fato de ser um número pertencente ao conjunto dos números inteiros;
- `number` — um número pertencente ao conjunto dos números reais. Segue a mesma regra de `integer`, sendo que `number` também contempla `integers`;
- `null` — um valor nulo;
- `object` — ou seja, um elemento que contém um conjunto de propriedades;
- `string` — ou seja, uma cadeia de caracteres.

Como pessoa é um objeto, podemos definir o tipo da seguinte forma:

```
{
  "title": "Pessoa",
  "type": "object"
}
```

Obviamente, um objeto precisa conter propriedades. Estas podem ser definidas a partir da propriedade `properties`. Por exemplo, se quisermos definir uma propriedade `nome` para uma pessoa, podemos utilizar o seguinte formato:

```
{
  "title": "Pessoa",
  "type": "object",
  "properties": {
    "nome": {
      "type": "string"
    }
  }
}
```

Note que, definida a propriedade `nome`, o conteúdo deste passa

a ter o mesmo formato da raiz `pessoa`, ou seja, existe uma recursividade. É possível, portanto, definir um atributo `title` dentro de `nome`, definir `nome` como um objeto com subpropriedades etc.

Assim sendo, podemos, portanto, definir um elemento `endereco` dentro de `pessoa`, que também será um objeto. Dessa forma, temos:

```
{  
    "title": "Pessoa",  
    "type": "object",  
    "properties": {  
        "nome": {  
            "type": "string"  
        },  
        "endereco": {  
            "type": "object",  
            "properties": {  
                "cep": {  
                    "type": "string"  
                },  
                "logradouro": {  
                    "type": "string"  
                }  
            }  
        }  
    }  
}
```

Finalmente, vale a pena observar que, por padrão, o JSON Schema não o limita aos formatos predefinidos (ao contrário do XML Schema, que apenas se atém ao definido). Por exemplo, de acordo com o JSON Schema anterior, o seguinte JSON seria considerado válido:

```
{  
    "nome": "Alexandre",  
    "cpf": "123.456.789-09"  
}
```

Note que, ainda que o campo `cpf` não esteja descrito, a presença dele não invalida o JSON, tornando os campos meramente

descritivos (ou seja, encarados como "esperados"). Caso este comportamento não seja desejável, é possível definir o atributo `additionalProperties` (que é do tipo booleano) como `false`, assim:

```
{  
    "title": "Pessoa",  
    "type": "object",  
    "additionalProperties": false,  
    "properties": {  
        "nome": {  
            "type": "string"  
        },  
        "endereco": {  
            "type": "object",  
            "additionalProperties": false,  
            "properties": {  
                "cep": {  
                    "type": "string"  
                },  
                "logradouro": {  
                    "type": "string"  
                }  
            }  
        }  
    }  
}
```

Desta forma, o JSON descrito seria considerado inválido.

Para realizar os testes de validação JSON, é possível usar o programa disponível em <https://github.com/fge/json-schema-validator>, ou ainda, utilizar sua plataforma online, disponível até a data da escrita deste livro em <http://json-schema-validator.herokuapp.com/>.

Para gerar código a partir deste JSON Schema, é possível usar um programa disponível em <https://github.com/joelittlejohn/jsonschema2pojo>. Este programa está disponível, até a data de escrita deste livro, sob os formatos online (em <http://www.jsonschema2pojo.org/>), como plugin Maven, plugin Gradle, tarefa Ant, um programa de linha de

comando ou uma API Java.

Se preferir, pode configurar no Maven com a seguinte instrução:

```
<build>
  <plugins>
    <plugin>
      <groupId>com.googlecode.jsonschema2pojo</groupId>
      <artifactId>jsonschema2pojo-maven-plugin</artifactId>
      <version>0.3.7</version>
      <configuration>
        <sourceDirectory>
          ${basedir}/src/main/resources/schema
        </sourceDirectory>
        <outputDirectory>
          ${basedir}/src/main/java
        </outputDirectory>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>generate</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Além disso, também é preciso configurar dependências com os artefatos `common-lang` e `jackson-databind`, que serão utilizados nas classes geradas. O XML usado por mim para configuração destas dependências foi:

```
<dependency>
  <groupId>commons-lang</groupId>
  <artifactId>commons-lang</artifactId>
  <version>2.4</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.0.0</version>
</dependency>
```

Desta forma, basta executar o comando `mvn generate-`

`sources` para que as classes `Pessoa` (cujo nome é gerado a partir do nome do arquivo — no meu caso, `Pessoa.json`) e `Endereco` (nome definido a partir da propriedade `endereco`) sejam geradas.

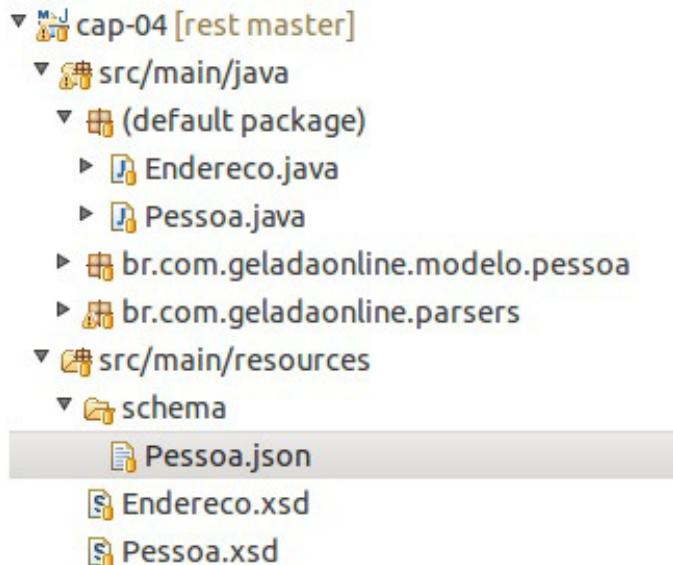


Figura 4.4: Classes geradas pelo plugin do Maven para JSON Schema

4.9 CONCLUSÃO

Neste capítulo, você conheceu mais a respeito dos formatos mais usados em REST, ou seja, XML e JSON, e também mais a respeito do ecossistema que os cerca, ou seja, no caso do XML, XML Schemas e JAXB. No caso de JSON, os JSON Schemas e as ferramentas habilitadas a trabalharem com este formato.

Deste momento em diante, você já tem em mãos os meios que serão utilizados para construção de serviços REST, ou seja, você já conhece os princípios de REST e já sabe usar os tipos de dados disponíveis para o seu uso. Mas ainda falta o primordial: *como combinar tudo isso?*

No próximo capítulo, você verá como combinar essa técnica via utilização de *servlets* Java. Mais adiante, você verá como utilizar APIs ainda melhores. Vamos em frente?

CAPÍTULO 5

IMPLEMENTANDO SERVIÇOS REST EM JAVA COM SERVLETS

"O único homem que não erra é aquele que nunca fez nada." –
Franklin Roosevelt

Agora que você já conhece os conceitos onde REST está envolvido, está na hora de ver como implementar isto em um sistema real. Como já é praxe, vou começar pela forma mais simples e, depois, partiremos para a implementação usando APIs próprias para construção de serviços REST.

5.1 UMA IMPLEMENTAÇÃO COM SERVLETS

O leitor que já tem experiência com *servlets* Java provavelmente já imaginou, até aqui, uma implementação com esta tecnologia. Mesmo se for este o seu caso, leia esta seção até o fim – pode lhe trazer informações úteis mesmo a respeito do funcionamento do mecanismo da API REST de Java.

Se não for o caso, saiba que a especificação Java EE possui definido o uso de servlets, ou seja, trechos de código específicos para serem executados no lado do servidor. Os servlets são, por natureza, construídos para atenderem a qualquer protocolo de aplicação que seja transmitido pela rede. No entanto, existe uma extensão deles

para trabalhar especificamente com o protocolo HTTP, que é justamente o que precisamos.

Esta extensão é a classe `javax.servlet.http.HttpServlet`, que possui os métodos conhecidos como `doXXX` (onde XXX é o nome de um método HTTP). Por exemplo, para atender a uma requisição que use o método `GET`, pode-se estender a classe `HttpServlet` e implementar o método `doGet`, assim:

```
package br.com.geladaonline.servlets;

import javax.servlet.http.*;

public class CervejaServlet extends HttpServlet {

    public void doGet (HttpServletRequest req,
                      HttpServletResponse resp) throws ServletException,
                      java.io.IOException {
        // Coloque aqui a implementação do seu código
    }
}
```

Este método vai carregar, então, a implementação do que se deseja para um serviço REST. Lembre-se de que o método `GET` é usado para realizar buscas no lado do servidor. Sendo este um servlet que busca dados de cervejas, deve-se implementá-lo de maneira a realizar esta ação.

Vamos utilizar aqui um conceito presente em *Domain-Driven Design*, que é o de **repositório**: uma classe de negócio especializada no armazenamento de entidades de negócio.

O QUE É DOMAIN-DRIVEN DESIGN?

Domain-Driven Design significa, em uma tradução livre, design orientado ao domínio. Isto quer dizer, em uma simplificação grosseira, que as classes que você construir devem sempre atender ao máximo de objetivos de negócio possível. Isso pode ser traduzido, por exemplo, em inserção de comportamento de negócio nas entidades que serão persistidas no banco de dados, o que traz ganhos em termos de coesão dos objetos.

Para saber mais, sugiro ler o livro de Eric Evans (2003), *Domain-Driven Design – Atacando as complexidades na criação do software*.

Vamos começar, então, pela definição da entidade que desejamos buscar, ou seja, uma cerveja:

```
package br.com.geladaonline.model;

public class Cerveja {

    private String nome;
    private String descricao;
    private String cervejaria;
    private Tipo tipo;

    public enum Tipo {
        LAGER, PILSEN, PALE_ALE, INDIAN_PALE_ALE, WEIZEN;
    }
}
```

A seguir, vamos realizar a definição do nosso repositório de cervejas (ou seja, um estoque de cervejas):

```
package br.com.geladaonline.model;

import java.util.*;
```

```

public class Estoque {
    private Collection<Cerveja> cervejas = new ArrayList<>();

    public Collection<Cerveja> listarCervejas() {
        return new ArrayList<>(this.cervejas);
    }

    public void adicionarCerveja (Cerveja cerveja) {
        this.cervejas.add(cerveja);
    }
}

```

Vamos incrementar o nosso estoque para começar com algumas cervejas (afinal de contas, ninguém gosta de um estoque vazio). Para isso, vamos modificar a classe `Cerveja` para criar um construtor para armazenar os parâmetros e, depois, modificar o construtor de estoque para criar essas cervejas. Então, a classe `Cerveja` fica assim:

```

public class Cerveja {

    private String nome;
    private String descricao;
    private String cervejaria;
    private Tipo tipo;

    public Cerveja(String nome, String descricao,
                   String cervejaria, Tipo tipo) {
        this.nome = nome;
        this.descricao = descricao;
        this.cervejaria = cervejaria;
        this.tipo = tipo;
    }

    // restante do código
}

```

E a classe `Estoque` fica assim:

```

public class Estoque {
    private Collection<Cerveja> cervejas = new ArrayList<>();

    public Estoque() {
        Cerveja primeiraCerveja = new Cerveja("Stella Artois",

```

```

        "A cerveja belga mais francesa do mundo :)",
        "Artois",
        Cerveja.Tipo.LAGER);
Cerveja segundaCerveja = new Cerveja("Erdinger Weissbier",
        "Cerveja de trigo alemã",
        "Erdinger Weissbräu",
        Cerveja.Tipo.WEIZEN);
this.cervejas.add(primeiraCerveja);
this.cervejas.add(segundaCerveja);
}

//restante do código
}

```

Assim, utilizamos a classe `Estoque` no nosso servlet de cervejas:

```

public class CervejaServlet extends HttpServlet {

    private Estoque estoque = new Estoque();

    public void doGet (HttpServletRequest req,
                      HttpServletResponse resp) throws ServletException,
                      java.io.IOException {
        // Coloque aqui a implementação do seu código
    }
}

```

Finalmente, para facilitar a nossa vida no quesito descritividade da cerveja, vamos sobrescrever o método `toString` da cerveja:

```

public class Cerveja {
    // o código já mostrado anteriormente

    public String toString() {
        return this.nome + " - " + this.descricao;
    }
}

```

Agora, podemos usar este conjunto de mecanismos para listar nossas cervejas a partir do nosso servlet. Para isso, vamos utilizar o mecanismo de impressão dos servlets, que é a obtenção de um `java.io.PrintWriter` a partir da classe `javax.servlet.http.HttpServletResponse`. Essa obtenção é

feita a partir do método `getWriter` :

```
package br.com.geladaonline.servlets;

import javax.servlet.http.*;
import java.io.*;
import br.com.geladaonline.model.*;

public class CervejaServlet extends HttpServlet {

    private Estoque estoque = new Estoque();

    public void doGet (HttpServletRequest req,
                      HttpServletResponse resp) throws ServletException,
                      java.io.IOException {
        PrintWriter out = resp.getWriter();
        Collection<Cerveja> cervejas = estoque.listarCervejas();
        for (Cerveja cerveja : cervejas) {
            out.print(cerveja);
        }
    }
}
```

Resta, agora, realizar o mapeamento deste servlet de maneira adequada. Seguindo os preceitos de REST, o ideal seria mapeá-lo para a URL `/cerveja` ou `/cervejas`, já que a nossa necessidade é buscar os dados das cervejas presentes no sistema.

Pela especificação 3.0 de servlets, é possível usar a anotação `javax.servlet.annotation.WebServlet` para descrevê-los perante o contêiner, criando o mapeamento adequado. Portanto, nossa classe mapeada pode ficar da seguinte forma:

```
// Declaração de package e imports

@WebServlet(value = "/cervejas/*")
public class CervejaServlet extends HttpServlet {
    // código
}
```

DICA DE USO

Caso tenha dificuldades em conferir o funcionamento deste código, basta ir ao repositório de código-fonte do livro, em <https://github.com/alesaudate/rest>, baixar o código e testá-lo em seu ambiente.

Finalmente, para conferir o funcionamento do código, basta abrir com o navegador o endereço <http://localhost:8080/cervejaria/cervejas>:

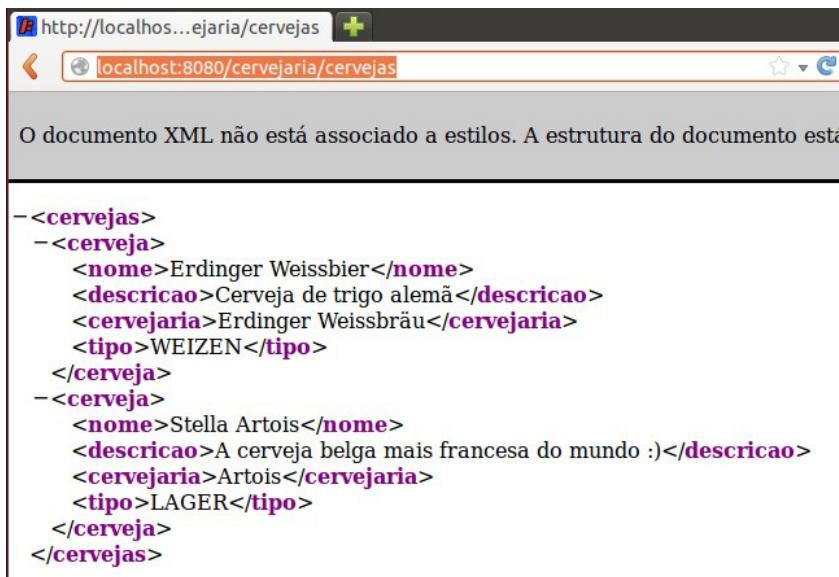


Figura 5.1: Teste no browser

No entanto, note que algo está errado. Para testar o serviço de uma maneira mais cômoda, sugiro a instalação do *plugin Poster*, para o navegador Mozilla Firefox, disponível em <https://addons.mozilla.org/pt-br/firefox/addon/poster/>.

USO OUTRO NAVEGADOR, O QUE FAÇO?

O *plugin* também está disponível para o Google Chrome, em <http://tinyurl.com/chrome-poster>. Não se preocupe, o uso nos dois navegadores é realizado da mesma forma.

Uma vez instalado o *plugin* no seu Firefox, basta abri-lo apontando para o menu **Ferramentas** (ou *Tools*, dependendo do idioma do seu navegador) e clicando em **Poster** :

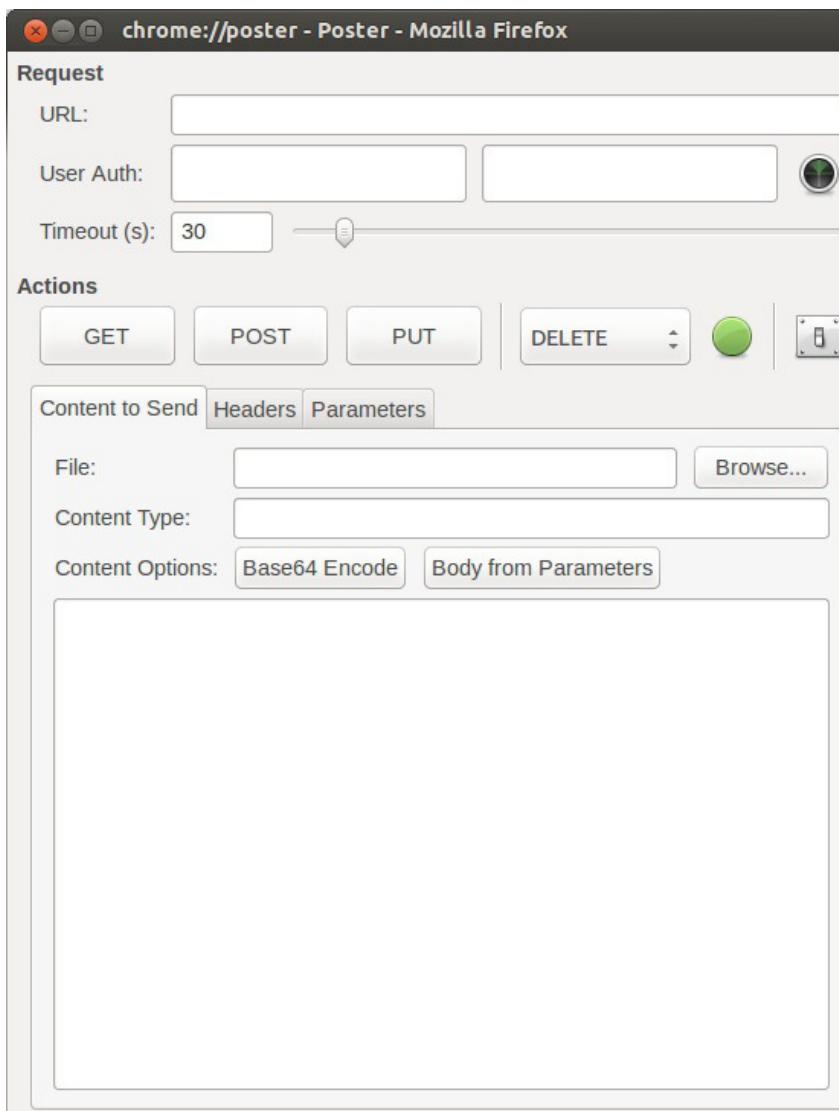


Figura 5.2: Poster, um plugin para o Firefox

Com o **Poster**, temos uma interface gráfica para testar os nossos serviços!

Na caixa **URL**, insira a URL do servlet de cervejas, ou seja, <http://localhost:8080/cervejaria/cervejas>. Na sequência, clique no

botão GET . O resultado deve aparecer de maneira idêntica à maneira como apareceu no navegador.

Porém, algo não está certo. O cabeçalho Content-Type não está presente, ou seja, nem sequer existe uma definição do tipo de dados. Além disso, o resultado foi impresso como texto puro – ou seja, o cliente não terá meios para identificar os tipos de dados trafegados etc.

Ou seja, nossa primeira missão é ajustar o tipo de dados. Para representar os dados, podemos utilizar os *MIME Types* text/xml ou application/xml para XML, e application/json para JSON. Você viu no capítulo *Tipos de dados* como manipular estes tipos.

Assim, suponha que o retorno desejado dessa listagem seja um conteúdo XML. Duas coisas são necessárias: criar um elemento raiz para os elementos e realizar as adaptações para trafegar estes dados no formato desejado. Para termos o menor impacto possível, portanto, temos como opção mais fácil utilizar o JAXB: a API já vem embutida nas implementações da *Virtual Machine Java* e também oferece compatibilidade com certos mecanismos geradores de JSON.

Assim, vamos criar uma classe que encapsule as outras cervejas, denominada Cervejas . Esta classe será apartada do modelo, e ficará localizada no pacote br.com.geladaonline.model.rest . Terá o seguinte código:

```
//Declaração de pacote e imports

@XmlRootElement
public class Cervejas {

    private List<Cerveja> cervejas = new ArrayList<>();

    @XmlElement(name="cerveja")
```

```

public List<Cerveja> getCervejas() {
    return cervejas;
}

public void setCervejas(List<Cerveja> cervejas) {
    this.cervejas = cervejas;
}
}

```

Como você pode observar, é apenas uma cápsula para cervejas. Ela tem dois motivos importantes para existir:

- Ela será usada para encapsular os outros elementos;
- Futuramente, usaremos essa classe para adicionar HATEOAS.

Agora, é hora de modificar a implementação do nosso servlet para que ele seja capaz de atender aos clientes. Para que isso aconteça, basta implementar novamente o método `doGet` no servlet, com o código de escrita de XML visto no capítulo *Tipos de dados*:

```

//declaração de pacotes, imports e da classe

private static JAXBContext context;

static {
    try {
        context = JAXBContext.newInstance(Cervejas.class);
    } catch (JAXBException e) {
        throw new RuntimeException(e);
    }
}

protected void doGet(HttpServletRequest req,
HttpServletResponse resp) throws ServletException,
IOException {

try {
    Marshaller marshaller = context.createMarshaller();
    resp.setContentType("application/xml;charset=UTF-8");
    PrintWriter out = resp.getWriter();
}

```

```
Cervejas cervejas = new Cervejas();
cervejas.setCervejas(new ArrayList<>(estoque.
                                         listarCervejas()));
marshaller.marshal(cervejas, out);

} catch (Exception e) {
    resp.sendError(500, e.getMessage());
}
}
```

PREVENINDO PROBLEMAS

Nesse código, note que a definição do `Content-Type` acompanha a definição do `charset`. Caso não seja definido, o contêiner (no meu caso, o `Jetty`) normalmente seleciona um predefinido. Observe que a ordem em que as coisas são feitas também influencia esta definição – no caso do `Jetty`, se este ajuste for feito depois da invocação a `resp.getWriter`, o container vai ignorar a definição do `charset`.

Observe que a `engine` do JAXB deposita o XML gerado diretamente na saída oferecida pela API de servlets. Para testar este código, recorremos novamente ao `Poster`:

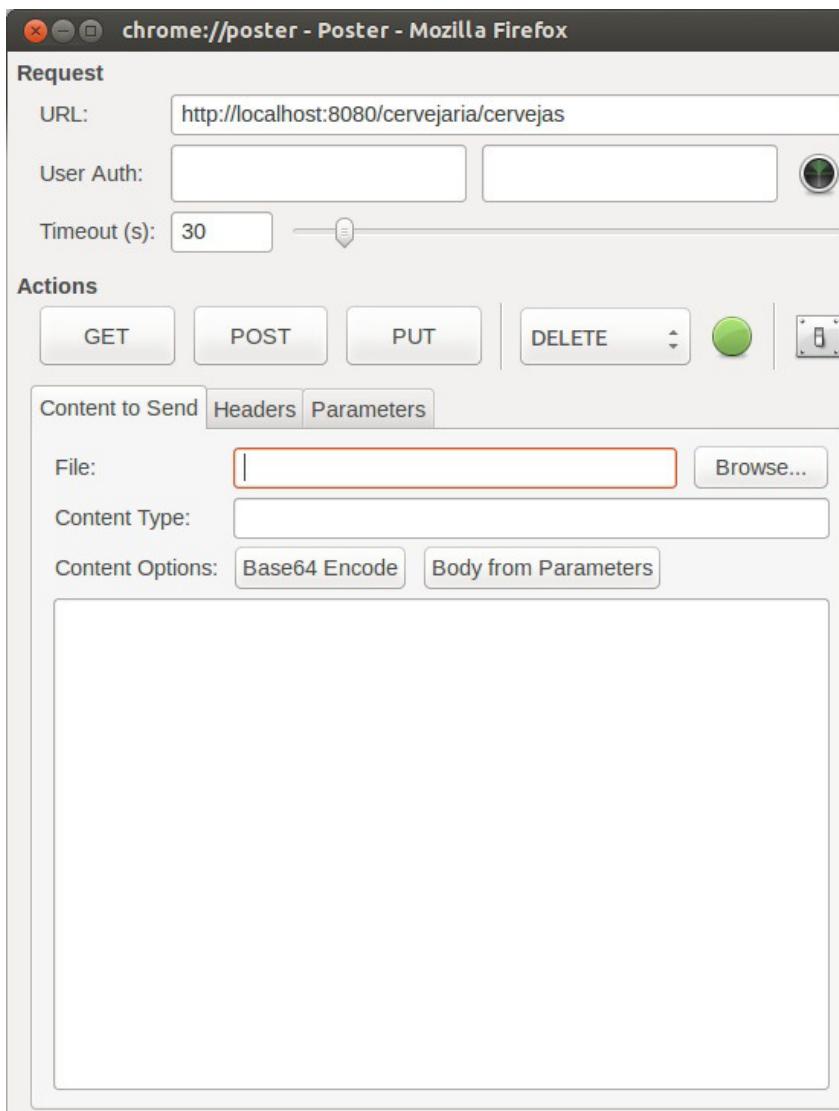


Figura 5.3: Enviando uma requisição de teste com o poster

A resposta será a seguinte:

GET on http://localhost:8080/cervejaria/cervejas

Status: 200 OK

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><cervejas>
<cerveja><nome>Erdinger Weissbier</nome><descricao>Cerveja de trigo  
alemã</descricao><cervejaria>Erdinger Weissbräu</cervejaria>
<tipo>WEIZEN</tipo></cerveja><cerveja><nome>Stella Artoois</nome>
<descricao>A cerveja belga mais francesa do mundo :)</descricao>
<cervejaria>Artoois</cervejaria><tipo>LAGER</tipo></cerveja>
</cervejas>
```

Headers:

Content-Type	application/xml; charset=UTF-8
Transfer-Encoding	chunked
Server	Jetty(8.1.9.v20130131)

[Close](#)

Figura 5.4: Resposta teste poster

5.2 IMPLEMENTANDO NEGOCIAÇÃO DE CONTEÚDO

No exemplo anterior, a URL /cervejas está habilitada a enviar para o cliente um conteúdo apenas em formato XML. Mas o que acontece se quisermos enviar dados de outras maneiras? Digamos que eu queira enviar dados como JSON, também, o que faço?

A saída não é criar novas URLs, mas sim, implementar uma técnica conhecida como **negociação de conteúdo**. Nela, o cliente diz para o servidor que tipo de dados deseja (por meio do cabeçalho `Accept`) e, então, o servidor produz o tipo de conteúdo no formato desejado.

Para implementar negociação de conteúdo com servlets é muito fácil; basta invocar o método `HttpServletRequest.getHeader`, assim:

```
String acceptHeader = req.getHeader("Accept");
```

A partir de então, é preciso testar o resultado. Lembre-se (de acordo com o capítulo *O protocolo HTTP*) de que essa negociação pode ser potencialmente compilada. Para evitar um código demasiadamente complexo, vamos apenas trabalhar com a **existência** ou não de formatos esperados neste cabeçalho. Assim, testamos primeiro para a solicitação de XML.

Caso não haja XML na requisição, testamos para JSON e, se o teste falhar, devolvemos o código de erro 415 (ainda de acordo com o capítulo *O protocolo HTTP*, significando que foi solicitado um formato de dados que não é suportado). O código fica assim:

```
if (acceptHeader == null ||  
    acceptHeader.contains("application/xml")) {  
    escreveXML(req, resp);  
} else if (acceptHeader.contains("application/json")) {  
    escreveJSON(req, resp);  
} else {  
    // O header accept foi recebido com um valor não suportado  
    resp.sendError(415); // Formato não suportado  
}
```

Agora, resta escrever os métodos `escreveXML` e `escreveJSON`. A esta altura, isto não deve ser nenhum mistério para você; mas listo adiante caso você ainda tenha dúvidas a respeito:

```

private void escreveXML(HttpServletRequest req,
                        HttpServletResponse resp)
throws ServletException, IOException {

    Cervejas cervejas = new Cervejas();
    cervejas.setCervejas(new ArrayList<>(estoque
                                              .listarCervejas()));

    try {
        resp.setContentType("application/xml;charset=UTF-8");
        Marshaller marshaller = context.createMarshaller();
        marshaller.marshal(cervejas, resp.getWriter());

    } catch (JAXBException e) {
        resp.sendError(500); //Erro interno inesperado
    }
}

//Este código assume o Jettison como provedor de mapeamento JSON

private void escreveJSON(HttpServletRequest req,
                        HttpServletResponse resp)
throws ServletException, IOException {

    Cervejas cervejas = new Cervejas();
    cervejas.setCervejas(new ArrayList<>(estoque
                                              .listarCervejas()));

    try {
        resp.setContentType("application/json;charset=UTF-8");
        MappedNamespaceConvention con =
            new MappedNamespaceConvention();

        XMLStreamWriter xmlStreamWriter =
            new MappedXMLStreamWriter(con, resp.getWriter());

        Marshaller marshaller = context.createMarshaller();
        marshaller.marshal(objetoAEscrever, xmlStreamWriter);

    } catch (JAXBException e) {
        resp.sendError(500);
    }
}

```

5.3 IMPLEMENTANDO A BUSCA POR UMA CERVEJA ESPECÍFICA

Até aqui, você já viu como listar todas as cervejas cadastradas no sistema. Mas como buscar uma cerveja específica?

Para resolver esta questão, o recomendado é modelar a URL de maneira que, se o cliente fornecer o ID da cerveja na URL, apenas uma única é retornada. Por exemplo, se eu quiser buscar a Stella Artois, posso fazê-lo pela URL `/cervejas/Stella+Artois` (note que esta já está codificada para ser usada em um *browser*).

Para fazer isso, no entanto, é necessário realizar algumas modificações, a começar pela classe `Estoque`: esta deve ser modificada para encarar os nomes das cervejas como identificadores. Assim, substituímos a implementação da classe `Estoque` para armazenar as cervejas em um mapa, e não em uma lista:

```
private Map<String, Cerveja> cervejas = new HashMap<>();  
  
public Collection<Cerveja> listarCervejas() {  
    return new ArrayList<>(this.cervejas.values());  
}  
  
public void adicionarCerveja (Cerveja cerveja) {  
    this.cervejas.put(cerveja.getNome(), cerveja);  
}
```

Assim, podemos acrescentar um método nesta classe para recuperar as cervejas pelo nome:

```
public Cerveja recuperarCervejaPeloNome (String nome) {  
    return this.cervejas.get(nome);  
}
```

O próximo passo é incluir no *servlet* um método para extrair o identificador (ou seja, o nome da cerveja) da URL enviada pelo cliente. Para fazer isso, precisamos utilizar o método

```
HttpServletRequest.getRequestURI :
```

```
String requestUri = req.getRequestURI();
```

De posse da URI invocada, precisamos extrair o conteúdo que está depois da última barra. Mas esta URL pode ter vários formatos:

- /cervejas
- /cervejas/
- /cervejas/Stella
- /cervejas/Stella+Artois

Ou seja, precisamos de um método que seja inteligente o suficiente para extrair o que vem depois de /cervejas . Existem várias maneira de fazer isso. O método criado por mim foi dividir a *string* pelos separadores (ou seja, /) e, depois, iterar pela lista de pedaços até detectar o contexto do servlet, ou seja, cervejas . Uma vez localizado o contexto, ajusta uma variável booleana indicando que, caso haja qualquer coisa na URL depois deste pedaço, é um identificador. O código fica assim:

```
String[] pedacosDaUri = requestUri.split("//");

boolean contextoCervejasEncontrado = false;
for (String contexto : pedacosDaUri) {
    if (contexto.equals("cervejas")) {
        contextoCervejasEncontrado = true;
        continue; //Faz o loop avançar até o próximo
    }
    if (contextoCervejasEncontrado) {
        return contexto;
    }
}
```

No entanto, este código só leva em consideração o caso de o identificador existir. Para o caso de a URL estar codificada, ou seja, estar como Stella+Artois (que, decodificado, representa Stella Artois), o tratamento ainda não está implementado. Mas decodificar é fácil, basta utilizar a classe `java.net.URLDecoder` :

```

if (contextoCervejasEncontrado) {
    try {
        //Tenta decodificar usando o charset UTF-8
        return URLDecoder.decode(contexto, "UTF-8");
    } catch (UnsupportedEncodingException e) {
        //Caso o charset não seja encontrado, faz o melhor esforço
        return URLDecoder.decode(contexto);
    }
}

```

Resta apenas o caso onde o identificador não existe. Para isso, o mais elegante seria modelar uma exceção própria, algo como `RecursoSemIdentificadorException`. O código finalizado fica assim:

```

private String obtemIdentificador(HttpServletRequest req)
    throws RecursoSemIdentificadorException {
    String requestUri = req.getRequestURI();

    String[] pedacosDaUri = requestUri.split("/");
    boolean contextoCervejasEncontrado = false;
    for (String contexto : pedacosDaUri) {
        if (contexto.equals("cervejas")) {
            contextoCervejasEncontrado = true;
            continue;
        }
        if (contextoCervejasEncontrado) {
            try {
                return URLDecoder.decode(contexto, "UTF-8");
            } catch (UnsupportedEncodingException e) {
                return URLDecoder.decode(contexto);
            }
        }
    }
    throw new RecursoSemIdentificadorException
        ("Recurso sem identificador");
}

```

O próximo passo é alterar os métodos de busca para tratarem o envio tanto de uma `Cerveja` quanto da classe que pode conter todas, ou seja, `Cervejas`. Para isso, basta criar um método para realizar a localização. Como estas classes não possuem parentesco

entre elas, usamos a classe `Object` como retorno do método:

```
private Object
localizaObjetoASerEnviado(HttpServletRequest req) {
    Object objeto = null;

    try {
        String identificador = obtemIdentificador(req);
        objeto = estoque.recuperarCervejaPeloNome(identificador);
    }
    catch (RecursoSemIdentificadorException e) {
        Cervejas cervejas = new Cervejas();
        cervejas.setCervejas(new ArrayList<>(estoque
                                         .listarCervejas()));
        objeto = cervejas;
    }

    return objeto;
}
```

Finalmente, basta alterar os métodos de escrita para utilizarem este novo método. Observe que existe uma possibilidade de o objeto retornado ser nulo, ou seja, caso um identificador tenha sido fornecido, mas o objeto não tenha sido localizado no estoque. Assim, é necessário retornar o erro 404 (`Not Found`) caso o objeto seja nulo. O método de escrita de XML e o de JSON serão semelhantes:

```
private void
escreveXML(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {

    Object objetoAEscrever = localizaObjetoASerEnviado(req);

    if (objetoAEscrever == null) {
        resp.sendError(404); //objeto não encontrado
        return ;
    }

    try {
        resp.setContentType("application/xml; charset=UTF-8");
        Marshaller marshaller = context.createMarshaller();
        marshaller.marshal(objetoAEscrever, resp.getWriter());

    } catch (JAXBException e) {
```

```
        resp.sendError(500);
    }

}
```

MAS É POSSÍVEL ENVIAR UM OBJECT PARA O JAXB?

Se você teve uma dúvida em relação ao envio da classe para o método `marshal`, você está indo no caminho certo :). Na verdade, neste caso estamos confiando sempre que o objeto a ser enviado para este método é compatível com JAXB, e este fará a detecção em **tempo de execução**, e não em tempo de compilação. Desta forma, podemos passar qualquer objeto para o método `marshal` e, caso este não seja compatível, uma `JAXBException` será lançada.

Agora, basta testar. Use novamente o `Poster`, passando a URL <http://localhost:8080/cervejaria/cervejas/Stella+Artois> como parâmetro. O seguinte deverá ser retornado:

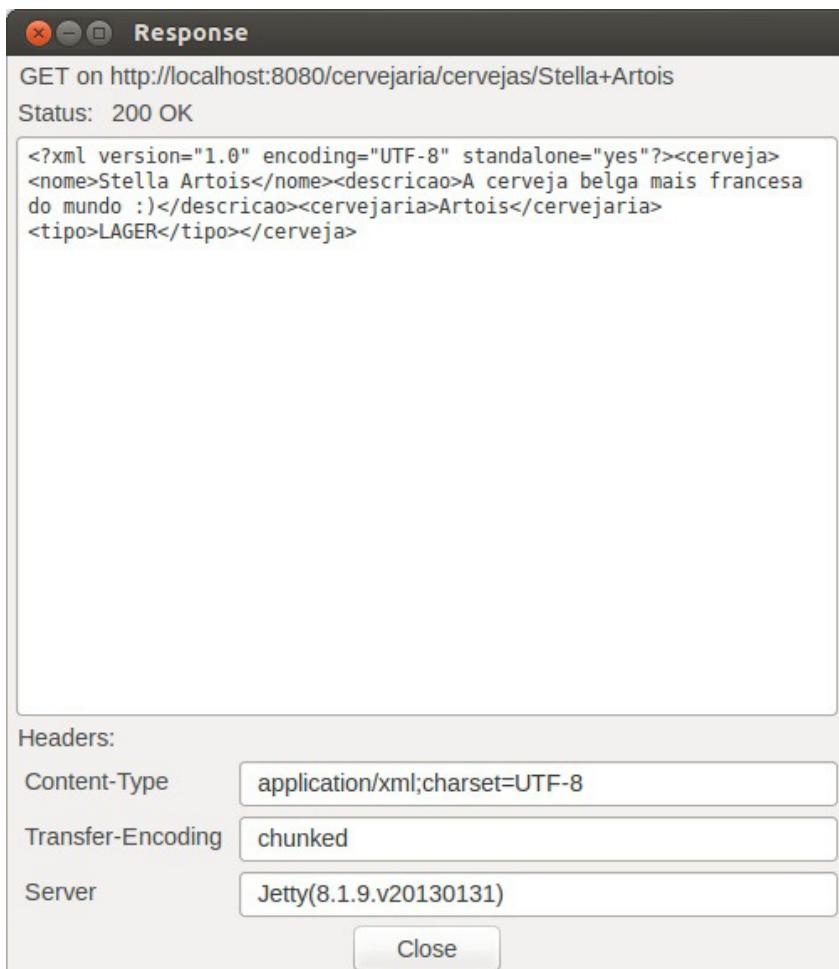


Figura 5.5: Retorno do teste do servlet de cervejas

5.4 IMPLEMENTANDO A CRIAÇÃO DE UM RECURSO

O seu servlet já busca recursos, mas será que está apto a realizar a criação destes?

Como você viu antes, tipicamente as criações de recursos são feitas por meio da utilização do método `POST`. Sendo assim, você

deve apenas implementar o método `doPost` no servlet:

```
protected void doPost(HttpServletRequest req,  
HttpServletResponse resp) throws ServletException, IOException
```

Para criação de recursos, você tem duas opções de modelagem: fazer com que o cliente passe um identificador, já criado por ele, ou criar o identificador no lado do servidor. Como nossa aplicação é de cervejas, faz sentido deixar que o cliente crie o identificador, já que os identificadores das cervejas são os nomes.

COMO ACONTECE EM APLICAÇÕES REAIS?

Na maior parte das aplicações REST, o servidor cria os identificadores, especialmente se o *back-end* da aplicação estiver implementado com um banco de dados relacional e os IDs são gerados por este banco. Mas você deve tomar o máximo de cuidado possível com esta abordagem, pois sua aplicação pode crescer e utilizar outros tipos de banco de dados. O ideal é que você sempre faça o projeto da sua API de maneira que você consiga modificar qualquer parte da sua infraestrutura.

Desta forma, a primeira coisa que você terá de implementar é a recuperação do identificador (isto deve ser trivial, já que você já tinha feito um método para isto quando criou o método de recuperação de cervejas). Caso o identificador não seja encontrado, envie um erro 400 Bad Request para o cliente, indicando que a requisição deve ser alterada e reenviada:

```
String identificador = null;  
try {  
    identificador = obtemIdentificador(req);  
} catch (RecursoSemIdentificadorException e) {  
    //Manda um erro 400
```

```
        resp.sendError(400, e.getMessage());
    }
```

O próximo tratamento a ser realizado é para o caso de a cerveja já existir; neste caso, um erro 409 `Conflict` deve ser lançado:

```
if (identificador != null &&
    estoque.recuperarCervejaPeloNome(identificador) != null){

    resp.sendError(409, "Já existe uma cerveja com esse nome");
    return;
}
```

Finalmente, você deve realizar a leitura do corpo da requisição e transformar em um objeto `Cerveja`. Para recuperar o corpo da requisição, basta executar o método `HttpServletRequest.getInputStream`. Felizmente, o restante do processo já é facilitado pelo JAXB:

```
Unmarshaller unmarshaller = context.createUnmarshaller();
Cerveja cerveja =
    (Cerveja)unmarshaller.unmarshal(req.getInputStream());
```

O próximo passo é garantir que a cerveja tenha o mesmo nome que o fornecido pelo identificador e, então, adicioná-la ao estoque:

```
cerveja.setNome(identificador);
estoque.adicionarCerveja(cerveja);
```

Também é necessário enviar para o cliente o código 201 `Created`, com o cabeçalho `Location`, indicando a URL onde a nova cerveja está disponível. Como nosso servlet já recebe a URL devidamente ajustada, basta ajustar a URL da requisição no cabeçalho `Location` através do método `HttpServletResponse.setHeader`:

```
String requestURI = req.getRequestURI();
resp.setHeader("Location", requestURI);
resp.setStatus(201);
```

Também é uma boa prática enviar de volta para o cliente o recurso criado, já que algo pode ter sido alterado na fase de criação

do recurso. Este processo será bastante facilitado por meio do uso do método `escreveXML`, criado anteriormente:

```
escreveXML(req, resp);
```

O método completo fica assim:

```
protected void doPost(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException{

    try {
        String identificador = null;
        try {
            identificador = obtemIdentificador(req);
        } catch (RecursoSemIdentificadorException e) {
            //Manda um erro 400 - Bad Request
            resp.sendError(400, e.getMessage());
        }

        if (identificador != null &&
estoque.recuperarCervejaPeloNome(identificador) != null) {
            resp.sendError(409,
                "Já existe uma cerveja com esse nome");
            return;
        }

        Unmarshaller unmarshaller = context.createUnmarshaller();
        Cerveja cerveja =
            (Cerveja)unmarshaller.unmarshal(req.getInputStream());
        cerveja.setNome(identificador);
        estoque.adicionarCerveja(cerveja);
        String requestURI = req.getRequestURI();
        resp.setHeader("Location", requestURI);
        resp.setStatus(201);
        escreveXML(req, resp);
    }
    catch (JAXBException e ) {
        resp.sendError(500, e.getMessage());
    }
}
```

Para testar este método, basta utilizar o `Poster` novamente, passando o XML da cerveja no corpo da requisição:

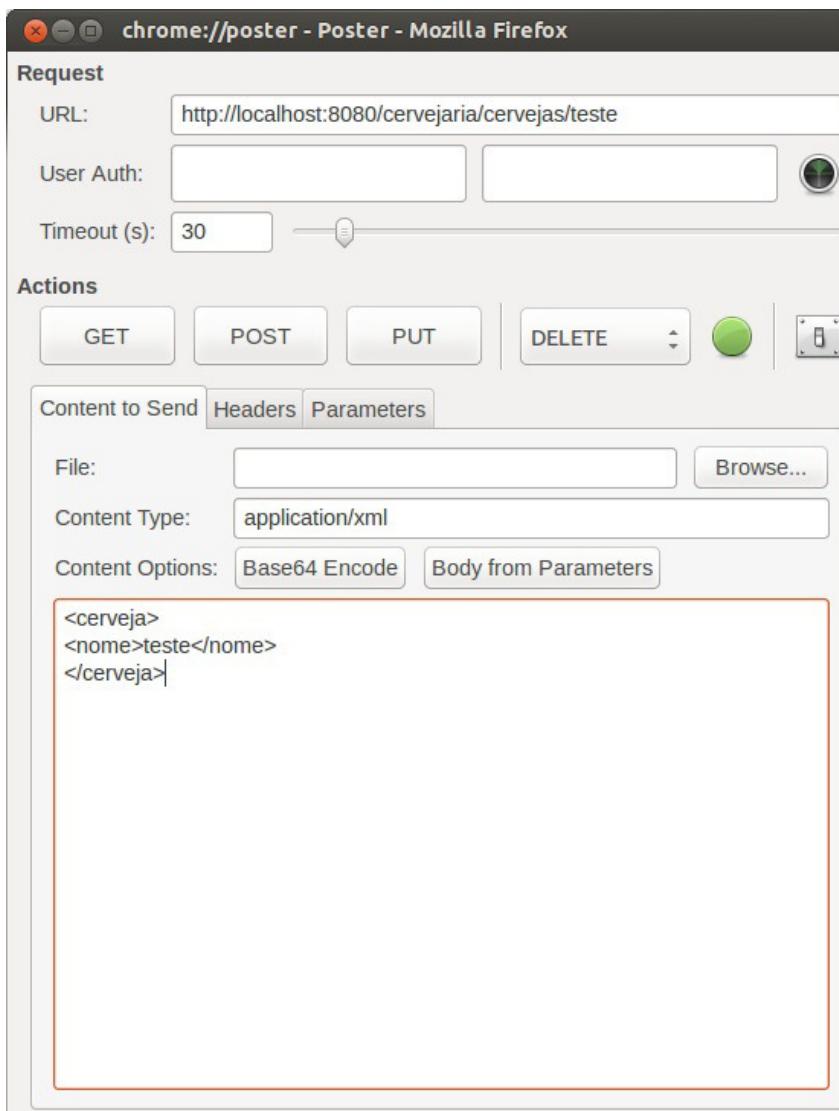


Figura 5.6: Exemplo de utilização do Poster para criação de um recurso

Caso o método tenha sido implementado corretamente, algo como o seguinte deve ser retornado:

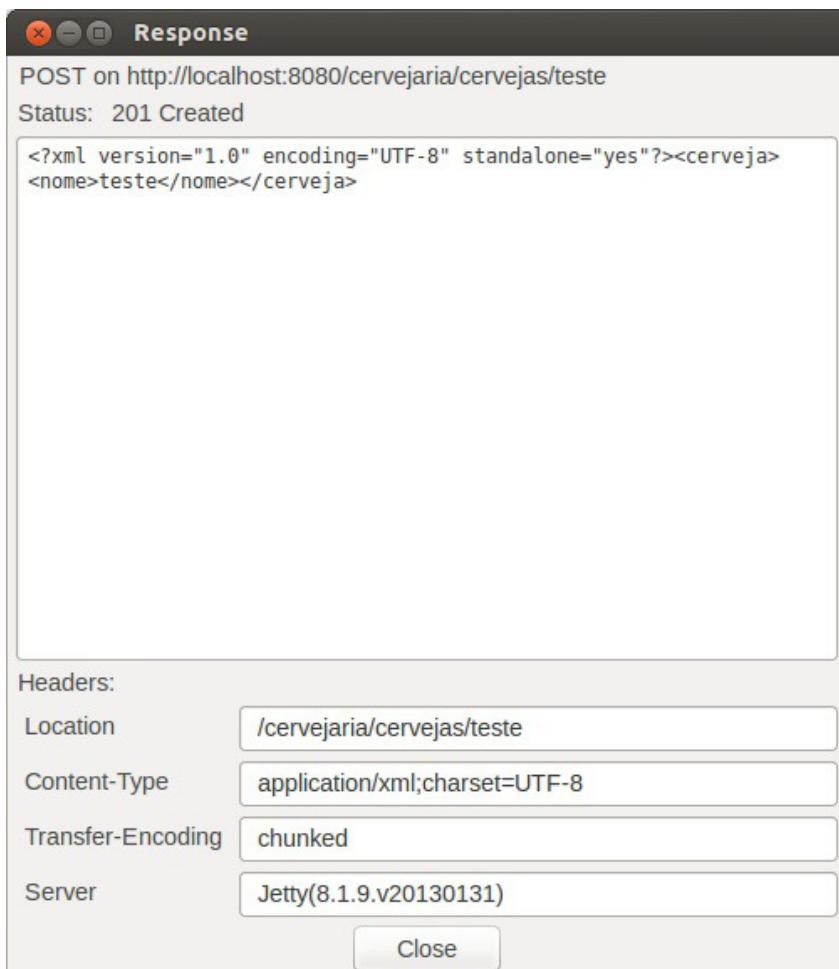


Figura 5.7: Resultado da criação do recurso

Além disso, note que, caso a mesma requisição seja submetida uma segunda vez, o código 409 deve ser retornado:

The screenshot shows a browser's developer tools interface, specifically the 'Response' tab. At the top, it says 'POST on http://localhost:8080/cervejaria/cervejas/teste'. Below that, 'Status: 409 J' is displayed. The main content area contains the following HTML code:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
<title>Error 409 Já existe uma cerveja com esse nome</title>
</head>
<body><h2>HTTP ERROR 409</h2>
<p>Problem accessing /cervejaria/cervejas/teste. Reason:<br/>
<pre>    Já existe uma cerveja com esse nome</pre></p><hr/>
<i><small>Powered by Jetty://</small><br/>
<br/>
```

Below the code, the 'Headers:' section is shown with the following entries:

Content-Type	text/html;charset=ISO-8859-1
Cache-Control	must-revalidate,no-cache,no-store
Content-Length	1442
Server	Jetty(8.1.9.v20130131)

A 'Close' button is located at the bottom right of the panel.

Figura 5.8: Resposta de erro indicando que o recurso já havia sido criado

5.5 IMPLEMENTANDO NEGOCIAÇÃO DE CONTEÚDO NO RECURSO

Para implementar negociação de conteúdo na criação do recurso, o cliente deve ajustar o cabeçalho `Content-Type` de maneira que indique o tipo do conteúdo presente na requisição. O servidor deve recuperar o conteúdo deste cabeçalho e tratá-lo. Para fazer isso com nosso servlet, basta executar o método

```
HttpServletRequest.getContentType :
```

```
String tipoDeConteudo = req.getContentType();
```

O próximo passo é incluir um `if / else if / else`. O primeiro fará o teste se o conteúdo for do tipo XML (`text/xml` ou `application/xml`); o segundo vai testar se o conteúdo for do tipo JSON (`application/json`); e o terceiro enviará um erro, informando que o tipo de dado não foi detectado.

No caso de o conteúdo ser XML, o código já preparado deve ser utilizado. No caso de ser JSON, você deve usar um código semelhante ao de escrita. No entanto, para ler JSON, devemos realizar a leitura do corpo da requisição manualmente – o modelo do Jettison não comporta a leitura de uma *stream*. Neste caso, utilizamos a classe `org.apache.commons.io.IOUtils` para realizar a leitura das linhas de dados presentes na *stream*:

```
List<String> lines = IOUtils.readLines(req.getInputStream());  
StringBuilder builder = new StringBuilder();  
for (String line : lines) {  
    builder.append(line);  
}
```

MEU PROJETO NÃO TEM A BIBLIOTECA DA APACHE, O QUE FAÇO?

Caso você utilize o Maven, basta adicionar a dependência:

```
<dependency>  
    <groupId>commons-io</groupId>  
    <artifactId>commons-io</artifactId>  
    <version>2.4</version>  
</dependency>
```

Se não for seu caso, você também pode fazer o *download* desta biblioteca no site: <http://commons.apache.org/proper/commons-io/>.

Na sequência, basta usar o método de leitura do Jettison para transformar o JSON em um objeto `Cerveja`:

```
MappedNamespaceConvention con = new MappedNamespaceConvention();
JSONObject jsonObject = new JSONObject(builder.toString());

XMLStreamReader xmlStreamReader =
    new MappedXMLStreamReader(jsonObject, con);

Unmarshaller unmarshaller = context.createUnmarshaller();
Cerveja cerveja =
    (Cerveja)unmarshaller.unmarshal(xmlStreamReader);
```

A partir daqui, o método já está praticamente pronto. Basta, agora, repetir os passos realizados para leitura de XML (não esquecendo de substituir o método final por `escreveJSON`, em vez de `escreveXML`):

```
cerveja.setNome(identificador);
estoque.adicionarCerveja(cerveja);
String requestURI = req.getRequestURI();
resp.setHeader("Location", requestURI);
resp.setStatus(201);

escreveJSON(req, resp);
```

Para testar, basta repetir o procedimento feito anteriormente, mas com uma requisição JSON:

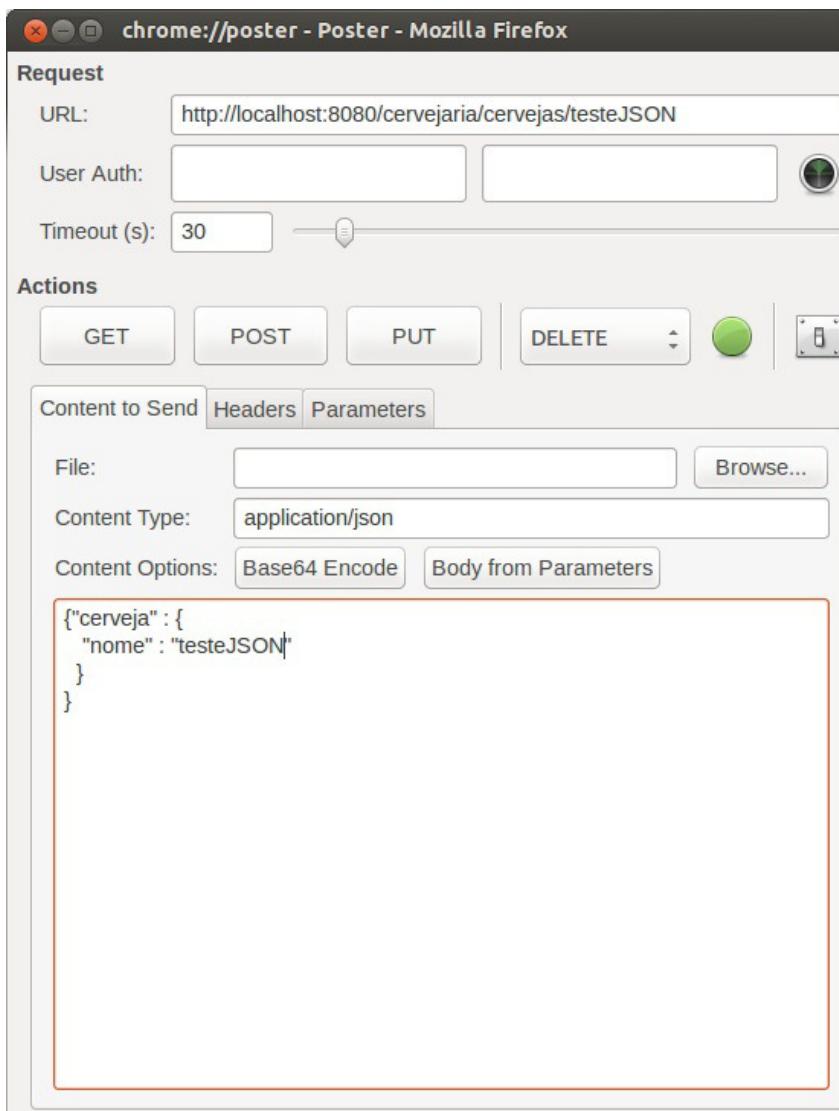


Figura 5.9: Requisição JSON

Se estiver correto, o resultado deve ser o seguinte:

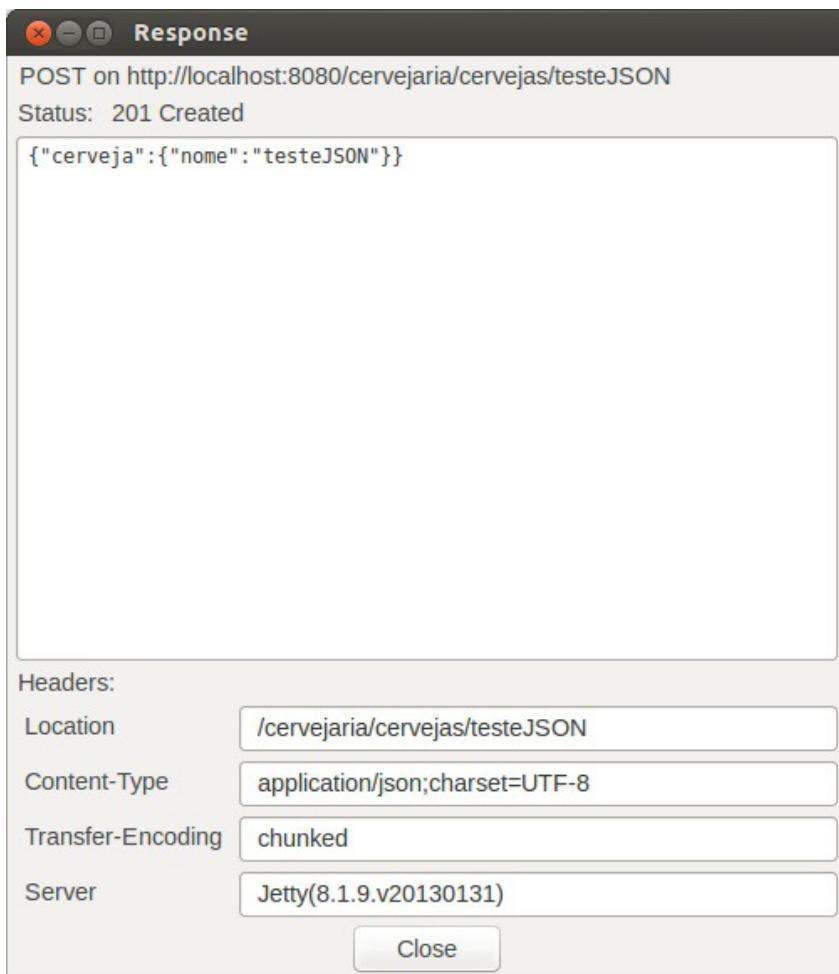


Figura 5.10: Resposta

Note que o recurso está presente no servidor independente do seu conteúdo, ou seja, se a requisição for feita uma segunda vez e não houver tipo de conteúdo definido, será retornado um XML.

5.6 CONCLUSÃO

Você viu neste capítulo como implementar serviços REST a

partir de servlets Java. Estes serviços fazem apenas a criação e recuperação de conteúdo (o restante são apenas variações destes dois tipos).

No entanto, você deve ter notado: é demasiado complexo realizar este processo, mesmo para um recurso simples. Você deve estar se perguntando: *existe alguma maneira, mais simples, de realizar este processo?* Sim, há!

Nos próximos capítulos, você verá como usar a especificação Java para criação de serviços REST, o JAX-RS. Além disso, você também verá como incluir links HATEOAS nos seus recursos de maneira eficiente, como testar esses serviços, tópicos sobre casos complexos de modelagem e muito mais. Vamos em frente?

CAPÍTULO 6

EXPLORE O JAX-RS

"O ignorante afirma, o sábio duvida, o sensato reflete." – Aristóteles

A linguagem Java possui, desde setembro de 2008, uma especificação própria para desenvolvimento de serviços em REST. Esta especificação é a JSR 311, popularmente conhecida como JAX-RS (*Java API for RESTful Web Services*).

Atualmente, esta especificação está em sua versão 2.0 (e o número da especificação foi alterado para 339). Esta será, portanto, a versão utilizada neste capítulo. Além disso, como uma especificação somente, o JAX-RS depende de implementações (assim como o JPA – Java Persistence API depende de implementações como o Hibernate). Será utilizada como implementação o Jersey, que é a implementação de referência (RI) da JAX-RS e que pode ser baixado em <https://jersey.java.net/download.html>.

6.1 CONFIGURANDO O JAX-RS

Existem diversas maneiras diferentes de configurar o JAX-RS, ainda mais após a adoção da especificação de servlets 3.0. Esta configuração do JAX-RS é totalmente baseada em conceitos de servlets, ou seja, basta realizar o mapeamento do servlet do Jersey e apontar para a classe de configuração. Este mapeamento pode ser feito da seguinte forma (na especificação servlets 3.0):

```

<web-app>

    <servlet>
        <servlet-name>
            br.com.geladaonline.services.ApplicationJAXRS
        </servlet-name>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>
            br.com.geladaonline.services.ApplicationJAXRS
        </servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>

```

O equivalente em código em *containers* que não implementam a especificação de servlets 3.0 seria o seguinte:

```

<web-app>
    <servlet>
        <servlet-name>Jersey Servlet</servlet-name>
        <servlet-class>
            org.glassfish.jersey.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>
                br.com.geladaonline.services.ApplicationJAXRS
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>Jersey Servlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>

```

Note o parâmetro `javax.ws.rs.Application`. Este aponta para uma classe que vai conter a configuração da aplicação JAX-RS. Esta classe deve estender a classe `javax.ws.rs.core.Application`:

```
package br.com.geladaonline.services;

import javax.ws.rs.core.Application;

public class ApplicationJAXRS extends Application{

}
```

Para realizar a configuração da aplicação, basta utilizar um dos métodos presentes na classe `Application`. Vamos voltar à configuração desta classe posteriormente. Antes, vamos criar nosso primeiro serviço JAX-RS para checar o funcionamento.

6.2 CRIANDO O PRIMEIRO SERVIÇO JAX-RS

Para criar um serviço JAX-RS, é necessário ter em mente os três pilares de um serviço REST:

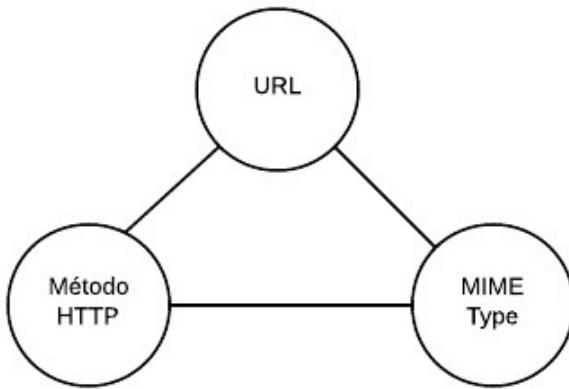


Figura 6.1: Pilares de um serviço REST

Perceba que, quando fizemos a definição do serviço com servlets, nossas principais tarefas eram realizar o *parse* da URL, do MIME Type e do método HTTP. Portanto, é necessário definir estes três pontos para que seja possível interagir com qualquer serviço

REST. Com JAX-RS, toda definição é feita utilizando anotações; estas vão ser vinculadas à *engine* JAX-RS para que seja possível invocar um método Java.

Para definir um serviço básico, podemos usar a seguinte classe:

```
package br.com.geladaonline.services;

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

@Path("/olaMundo")
public class TesteService {

    @GET
    @Produces("text/plain")
    public String dizOla() {
        return "Olá, mundo REST!";
    }
}
```

Note a presença das anotações `@Path` , `@GET` e `@Produces` . A primeira diz respeito ao mapeamento da URL que será utilizada; a segunda, o método HTTP (no caso, GET), e a terceira, o tipo de dados produzido pelo serviço (no caso, texto plano, `text/plain`). Não se preocupe se o uso destas anotações ficar muito abstrato agora, elas serão explicadas com mais detalhes adiante.

Para completar a configuração desta classe, basta sobrescrever o método `getClasses` na classe de configuração da nossa aplicação. Este método retorna um `java.util.Set` com as definições das classes de serviços. Portanto, a definição do método ficaria assim:

```
public class ApplicationJAXRS extends Application{

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<>();
        classes.add(TesteService.class);
    }
}
```

```
        return classes;
    }
}
```

Para testar, basta acessar a URL <http://localhost:8080/cervejaria/olaMundo>. Você deve ver o seguinte:



Figura 6.2: Primeiro serviço REST

6.3 O QUE ACONTECEU?

Antes de tudo, vamos observar o ciclo de vida do serviço. Ao realizar o mapeamento do servlet do Jersey, passando como parâmetro a classe `ApplicationJAXRS`, a engine do Jersey detectou que se tratava de uma classe que estendia `Application`. Ao realizar esta detecção, é possível instanciar a classe, e também identificar que esta possui os métodos que a classe `Application` possui. Estes métodos são:

- `getClasses` – para obter as classes dos serviços;
- `getSingletons` – para obter implementações de *features* do sistema;
- `getProperties` – para oferecer parâmetros de configuração extras.

Logo na inicialização do sistema, a engine do JAX-RS invoca os três métodos. A partir da obtenção das classes de serviço, cada invocação que for feita segue o "mapa" que foi previamente estabelecido. Uma vez que a engine detecta qual o método a ser

invocado (com base na URL, MIME Type e método HTTP), então instancia a classe e faz a invocação do método. Este mecanismo é seguido quando o escopo da classe é baseado na requisição – isto pode ser alterado, utilizando algumas anotações próprias do Jersey ou do Java EE 7.

6.4 DESENVOLVENDO OS MÉTODOS DE CONSULTA DE CERVEJAS

Vamos agora desenvolver o serviço de cervejas. Antes de fazer isso, no entanto, há uma melhoria que podemos fazer: para evitar ficar incluindo todo serviço novo que for criado no método `getClasses`, podemos usar uma propriedade do Jersey que faz com que a engine mapeie todos os serviços em um mesmo pacote, evitando essa alteração e eliminando a necessidade de sobrescrever o método `getClasses`. Essa propriedade é `jersey.config.server.provider.packages`. Para passar este parâmetro, basta sobrescrever o método `getProperties`:

```
public class ApplicationJAXRS extends Application{

    @Override
    public Map<String, Object> getProperties() {
        Map<String, Object> properties = new HashMap<>();
        properties.put("jersey.config.server.provider.packages",
                      "br.com.geladaonline.services");

        return properties;
    }
}
```

Note que, uma vez passado este parâmetro, todos os serviços presentes no pacote `br.com.geladaonline.services` serão automaticamente mapeados.

Agora, vamos desenvolver o nosso serviço de cervejas. Você já viu como mapear um serviço simples; o de cervejas seguirá a mesma

noção. Vamos recapitular, antes, o formato do nosso serviço: seguindo o conceito do tripé de serviços, a nossa URL deve ser `/cervejas`, e esse serviço deve utilizar o MIME Type `application/xml`. Este serviço deve ser, inicialmente, um simples CRUD de cervejas, ou seja, deve usar os métodos `GET`, `POST`, `PUT` e `DELETE`.

Assim sendo, vamos preparar a classe do serviço dessa forma:

```
package br.com.geladaonline.services;

import javax.ws.rs.*;
import javax.ws.rs.core.*;

@Path("/cervejas")
@Consumes({MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_XML})
public class CervejaService {

}
```

Para realizar a listagem de todas as cervejas, vamos reutilizar a classe `Cervejas` criada no capítulo *Implementando serviços REST em Java com Servlets*. Antes, vamos apenas acrescentar um construtor nesta classe para facilitar o uso:

```
public class Cervejas {

    private List<Cerveja> cervejas = new ArrayList<>();

    public Cervejas(List<Cerveja> cervejas) {
        this.cervejas = cervejas;
    }

    public Cervejas() {}

    //Restante do código
}
```

Agora, vamos à implementação do serviço:

```
package br.com.geladaonline.services;
```

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

```

import javax.ws.rs.*;
import javax.ws.rs.core.*;

import br.com.geladaonline.model.*;
import br.com.geladaonline.model.rest.Cervejas;

@Path("/cervejas")
@Consumes({MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_XML})
public class CervejaService {

    private static Estoque estoque = new Estoque();

    @GET
    public Cervejas listeTodasAsCervejas() {

        List<Cerveja> cervejas = estoque.listarCervejas();
        return new Cervejas(cervejas);
    }
}

```

Se você inicializar o serviço dessa forma, pode fazer o teste diretamente pelo browser:

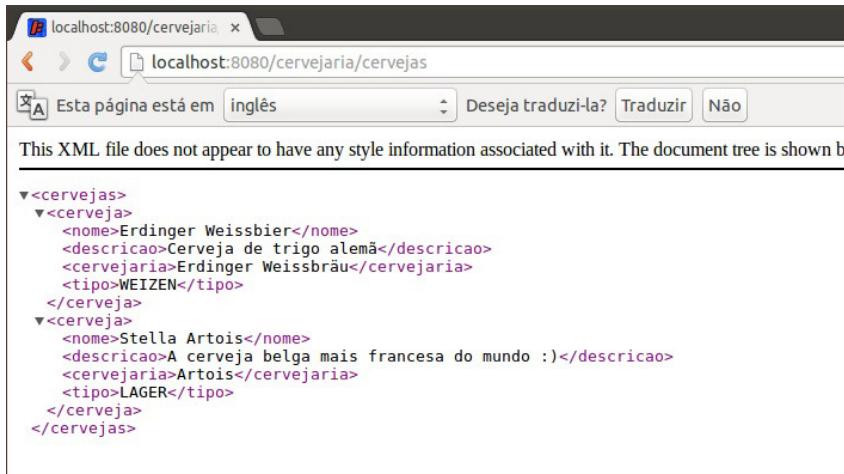


Figura 6.3: Teste do serviço pelo browser

Agora, precisamos realizar a localização de cervejas específicas pelo nome. Por exemplo, para que seja possível localizar a Stella

Artois, podemos acessar a URL `/cervejas/Stella Artois`. Note que o nome da cerveja deve ser parametrizável. Assim, vamos utilizar a anotação `@Path` diretamente no método, com um *placeholder*. *Placeholders* são delimitados usando chaves (`{` e `}`) diretamente na anotação, assim:

```
@GET  
@Path("{nome}")  
public Cerveja encontreCerveja
```

Note que é necessário fazer com que o código Java receba esse parâmetro. Porém, ele, após compilado, não mantém os nomes dos parâmetros de seus métodos. Para contornar essa limitação, usamos a anotação `@PathParam` diretamente no parâmetro, ligando o parâmetro ao *placeholder* na anotação `@Path`. Assim sendo, a assinatura desse método ficará:

```
@GET  
@Path("{nome}")  
public Cerveja encontreCerveja(  
    @PathParam("nome") String nomeDaCerveja) {
```

Finalmente, vamos à implementação. Note que, se a cerveja procurada não for encontrada, o código de status 404 deve ser retornado (de acordo com o que você viu no capítulo *O protocolo HTTP*). Isso pode (e deve) ser encarado de maneira semelhante ao lançamento de uma exceção Java. A especificação JAX-RS possui uma anotação especialmente preparada para este fim, que é a `javax.ws.rs.WebApplicationException`. Esta exceção recebe como parâmetro o código de erro HTTP (entre outros parâmetros). Assim, temos o seguinte código:

```
@GET  
@Path("{nome}")  
public Cerveja encontreCerveja(  
    @PathParam("nome") String nomeDaCerveja) {  
    Cerveja cerveja =  
        estoque.recuperarCervejaPeloNome(nomeDaCerveja);  
    if (cerveja != null)  
        return cerveja;
```

```
        throw new WebApplicationException(404);  
    }  

```

E QUANTO À ANOTAÇÃO DA CLASSE?

A anotação usada na própria definição da classe e a usada no método têm um efeito cumulativo, ou seja, as duas são utilizadas para compor uma URL. Ou seja, a URL de testes permanece sendo: /cervejas/{nome} .

Para testar, abra o browser e forneça a URL <http://localhost:8080/cervejaria/cervejas/Stella Artois>:

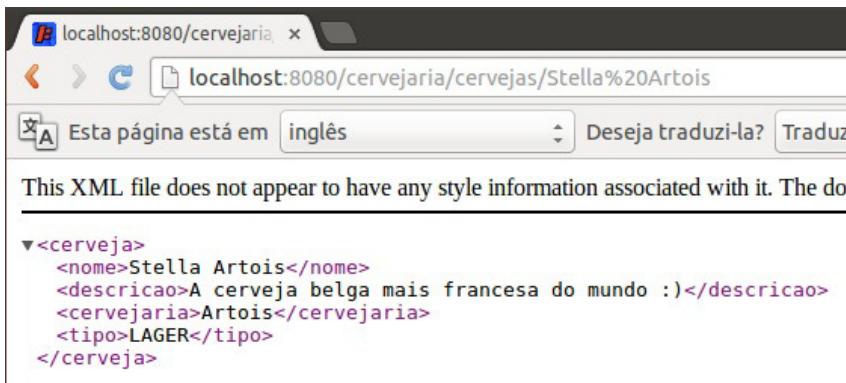


Figura 6.4: Teste de localização de uma cerveja específica

Agora, testamos com uma cerveja que não existe na base:

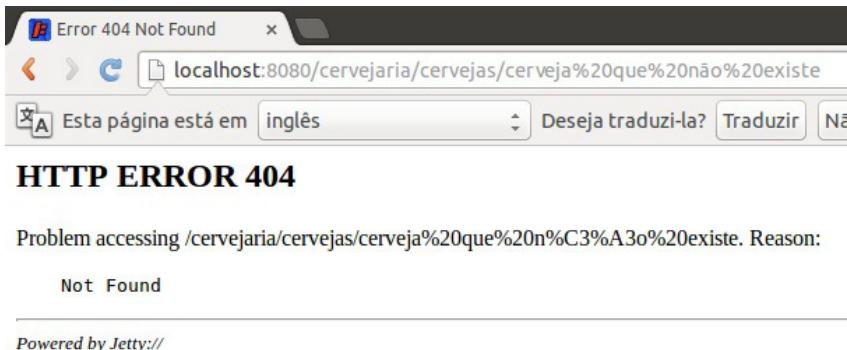


Figura 6.5: Teste de localização de uma cerveja que não existe no registro

Finalmente, falta um último toque neste código. Note que o código 404, no meio do código, não tem sentido caso seja lido por um leigo. Ele pode ser refatorado e trocado por uma constante, presente na classe `javax.ws.rs.core.Response.Status`. Assim sendo, esse código pode ser refatorado para:

```
throw new WebApplicationException(Status.NOT_FOUND);
```

6.5 IMPLEMENTANDO O MÉTODO DE CRIAÇÃO DE NOVAS CERVEJAS

Como você sabe, a criação de recursos em REST é feita pelo método `POST`, e deve retornar o código de status 201, em conjunto com o cabeçalho `Location`. Logo, preparamos o método utilizando a anotação `@POST`:

```
@POST  
public Cerveja criarCerveja(Cerveja cerveja) {  
    estoque.adicionarCerveja(cerveja);  
    return cerveja;  
}
```

Note que, neste caso, passamos a entidade `Cerveja` como parâmetro. O `parse` desta entidade será realizado de acordo com a

anotação `@Consumes`, ou seja, deve ser fornecido como parâmetro um XML que será transformado na entidade `Cerveja`.

No entanto, se você realizar o teste dessa forma, vai obter o seguinte resultado:

Response

POST on <http://localhost:8080/cervejaria/cervejas>

Status: 200 OK

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><cerveja>
<nome>Heineken</nome><descricao>Cerveja holandesa</descricao>
<cervejaria>Heineken</cervejaria><tipo>LAGER</tipo></cerveja>
```

Headers:

Content-Length	186
Content-Type	application/xml
Server	Jetty(8.1.9.v20130131)

[Close](#)

Figura 6.6: Resultado do teste de criação da nova cerveja

Ou seja, o código de status do retorno foi 200 em vez de 201. Para modificarmos esse resultado, é necessário customizar o

resultado por meio do uso da classe `javax.ws.rs.core.Response`. Esta pode ser colocada no retorno do método, e é configurada diretamente pelos métodos auxiliares da própria classe.

Antes disso, no entanto, é necessário configurar a URL de retorno do cabeçalho `Location`. A criação da URL é feita por meio da classe auxiliar `javax.ws.rs.core.UriBuilder`. Esta possui o método `fromPath`, para o qual pode ser fornecido um *template* (no mesmo molde das URLs que já criamos). Assim, para criar corretamente a URL da cerveja criada, basta ter algo como o seguinte:

```
java.net.URI uri =  
    UriBuilder.fromPath("cervejas/{nome}")  
        .build(cerveja.getNome());
```

Note que esta URL é criada apontando para o endereço onde a nova cerveja pode ser obtida.

Agora, faremos a criação da resposta. Para retornar o cabeçalho 201, utilizamos o método `created`, passando a URL que será retornada no cabeçalho `Location` como parâmetro. Além disso, também usamos o método `entity` para devolver o recurso criado:

```
Response.created(uri).entity(cerveja).build();
```

De acordo com o exposto no capítulo *O protocolo HTTP*, o método `POST` não é idempotente. Assim sendo, precisamos modificar a implementação para que o estoque rejeite criar cervejas que já existem. Portanto, podemos modificar o método `adicionarCerveja` para ficar assim:

```
public void adicionarCerveja(Cerveja cerveja) {  
    if (this.cervejas.containsKey(cerveja.getNome())) {  
        throw new CervejaJaExisteException();  
    }  
    this.cervejas.put(cerveja.getNome(), cerveja);  
}
```

Então, modificamos o código do nosso método de criação para lançar uma `WebApplicationException` com o código `409 Conflict`, indicando que houve um conflito na criação do recurso. O código do método completo fica assim:

```
@POST  
public Response criarCerveja(Cerveja cerveja) {  
    try {  
        estoque.adicionarCerveja(cerveja);  
    }  
    catch (CervejaJaExisteException e) {  
        throw new WebApplicationException(Status.CONFLICT);  
    }  
  
    URI uri =  
        UriBuilder.fromPath("cervejas/{nome}")  
            .build(cerveja.getNome());  
  
    return Response.created(uri).entity(cerveja).build();  
}
```

Assim, o resultado do teste fica assim:

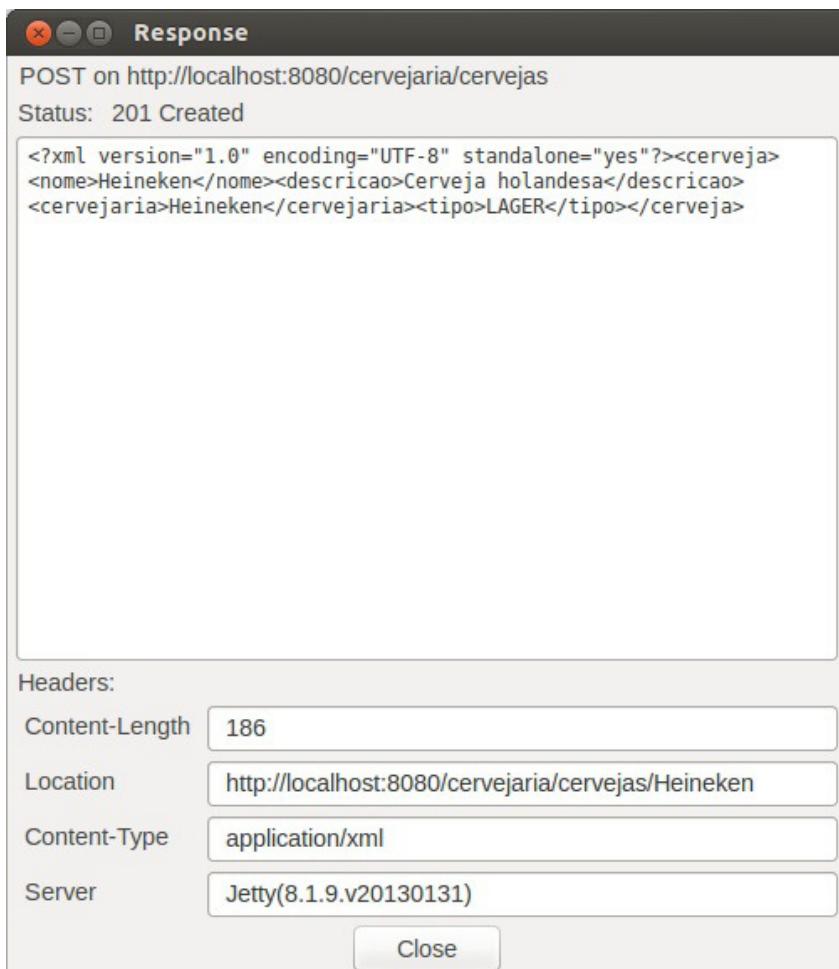


Figura 6.7: Resultado do teste da criação de recursos

6.6 MÉTODOS PARA ATUALIZAR E APAGAR RECURSOS

Finalmente, faltam os métodos para atualizar e apagar cervejas. Como você sabe, isso é feito pelos métodos `PUT` e `DELETE`, respectivamente. Também deve levar em consideração que ambos devem ser idempotentes (conforme visto no capítulo *O protocolo HTTP*).

Sabendo destas informações, o método de atualização fica assim:

```
@PUT  
@Path("{nome}")  
public void atualizarCerveja(@PathParam("nome") String nome,  
                           Cerveja cerveja) {  
    encontreCerveja(nome);  
    cerveja.setNome(nome);  
    estoque.atualizarCerveja(cerveja);  
}
```

Note que, caso a cerveja não exista, a invocação do método `encontreCerveja` vai provocar o lançamento de uma `WebApplicationException`, e o cliente receberá o código de status 404 como retorno.

Quanto ao método de exclusão, terá o seguinte formato:

```
@DELETE  
@Path("{nome}")  
public void apagarCerveja(@PathParam("nome") String nome) {  
    estoque.apagarCerveja(nome);  
}
```

Note que a idempotência está implementada em ambos os casos. No primeiro caso, se a mesma cerveja for submetida para atualização várias vezes, o resultado será sempre o mesmo. No segundo, se a cerveja não existir mais, a requisição será simplesmente ignorada. Em ambos os casos, o retorno será dado com o código 204 No Content (já que o retorno de ambos os métodos é `void`).

6.7 IMPLEMENTANDO LINKS HATEOAS

O último ponto que resta implementar para que nosso serviço de cervejas seja considerado verdadeiramente *RESTful* é a inclusão de links HATEOAS (conforme visto no capítulo *Conceitos de REST*). O JAX-RS 2 oferece um sistema pronto para uso de links HATEOAS por meio do uso da classe `javax.ws.rs.core.Link`.

Por exemplo, suponha que queiramos alterar o retorno da nossa consulta de cervejas para ficar da seguinte forma:

```
<cervejas>
    <link href="/cervejas/Erdinger%20Weissbier"
          title="Erdinger Weissbier" rel="cerveja"/>
    <link href="/cervejas/Stella%20Artois"
          title="Stella Artois" rel="cerveja"/>
</cervejas>
```

Neste caso, vamos modificar a classe `Cervejas` para que esse mapeamento seja possível. Conforme visto no capítulo *Tipos de dados*, vamos primeiro retirar o mapeamento das cervejas perante o JAXB. Isso é feito por meio do uso da anotação `@XmlTransient`:

```
@XmlTransient
public List<Cerveja> getCervejas() {
    return cervejas;
}
```

O próximo passo é incluir o mapeamento dos links. Conforme mencionado, o uso da classe `Link` vai trazer o resultado desejado. Para isso, vamos construir os links pelo caminho para obtenção de cada cerveja, de acordo com o descrito na classe `CervejaService`. Assim, o caminho fica `cervejas/{nome}`. Observe que o *placeholder* é mantido; a engine do JAX-RS vai fazer a substituição do valor pelo fornecido na invocação do método `build`:

```
public List<Link> getLinks() {
    List<Link> links = new ArrayList<>();
    for (Cerveja cerveja : getCervejas()) {

        Link link = Link.fromPath("cervejas/{nome}")
            .build(cerveja.getNome());

        links.add(link);
    }

    return links;
}
```

No entanto, ao testar este código, você percebe algo errado no

retorno:

```
<cervejas>
  <link />
  <link />
</cervejas>
```

Por que isso aconteceu? A classe `Link` não é própria para funcionamento com o JAXB e, por si só, não é mapeada completamente para XML. A correção do problema é simples: precisamos informar ao JAXB que ele deverá usar um adaptador para conversão destes links. Felizmente, este adaptador já existe: é a classe `javax.ws.rs.core.Link.JaxbAdapter`.

O mecanismo que informa ao JAXB como usar o adaptador é a anotação

`javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter`, que pode ter seu uso estendido ao pacote todo, por meio do uso de um arquivo `package-info.java`:

```
@XmlJavaTypeAdapter(value=JaxbAdapter.class)
package br.com.geladaonline.model.rest;

import javax.ws.rs.core.Link.JaxbAdapter;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
```

Agora, o teste fica bem diferente:

```
<cervejas>
  <link href="cervejas/Erdinger%20Weissbier"/>
  <link href="cervejas/Stella%20Artois"/>
</cervejas>
```

Finalmente, para incluir os atributos `rel` e `title`, utilizamos os métodos de mesmo nome na classe `Link`:

```
@XmlElement(name="link")
public List<Link> getLinks() {
    List<Link> links = new ArrayList<>();
    for (Cerveja cerveja : getCervejas()) {

        Link link = Link.fromPath("cervejas/{nome}")
            .rel("cerveja")
```

```

        .title(cerveja.getNome())
        .build(cerveja.getNome());

    links.add(link);
}

return links;
}

```

Finalmente, o resultado fica assim:

```

<cervejas>
  <link href="cervejas/Erdinger%20Weissbier"
    title="Erdinger Weissbier" rel="cerveja"/>
  <link href="cervejas/Stella%20Artois"
    title="Stella Artois" rel="cerveja"/>
</cervejas>

```

6.8 IMPLEMENTANDO PAGINAÇÃO NA CONSULTA

Como você viu na seção *Boas práticas de passagem de parâmetros*, a melhor forma de passar parâmetros opcionais, como um número de paginação de um recurso, são os *query parameters*. Assim sendo, vamos ver nesta seção como criar paginação usando JAX-RS. Para isso, vamos primeiro visualizar como será nosso recurso.

A URL para paginação deverá ficar assim: `/cervejas?pagina=0`. Isto trará os primeiros 20 resultados. Se utilizarmos a URL `/cervejas?pagina=1`, serão mostrados os próximos 20 resultados, e assim por diante. Finalmente, caso o parâmetro não seja fornecido, o serviço trará como resultado o conteúdo da primeira página (ou seja, será equivalente a `/cervejas?pagina=0`).

Para que seja possível usar *query parameters*, basta utilizar a anotação `javax.ws.rs.QueryParam` diretamente em um parâmetro para o método Java, assim:

```
@GET  
public Cervejas  
    listeTodasAsCervejas(@QueryParam("pagina") int pagina)
```

Como se trata de um tipo primitivo Java, caso o valor da página não seja fornecido, ele assumirá o valor padrão (ou seja, 0). Caso contrário, assumirá o valor passado como parâmetro. Agora, só o que resta é modificar o código para paginar os resultados. Para isso, primeiro criamos um método novo na classe `Estoque`, que fará o cálculo de paginação:

```
public List<Cerveja> listarCervejas(  
    int numeroPagina, int tamanhoPagina) {  
    int indiceInicial = numeroPagina * tamanhoPagina;  
    int indiceFinal = indiceInicial + tamanhoPagina;  
  
    List<Cerveja> cervejas = listarCervejas();  
  
    if (cervejas.size() > indiceInicial) {  
        if (cervejas.size() > indiceFinal) {  
            cervejas =  
                cervejas.subList(indiceInicial, indiceFinal);  
        } else {  
            cervejas =  
                cervejas.subList(indiceInicial, cervejas.size());  
        }  
    } else {  
        cervejas = new ArrayList<>();  
    }  
    return cervejas;  
}
```

UMA IMPLEMENTAÇÃO REAL

Em uma implementação real, este código delegaria a questão da paginação para a infraestrutura (como o banco de dados, por exemplo).

Finalmente, alteramos a implementação do serviço em si para

trabalhar com esta questão:

```
private static final int TAMANHO_PAGINA = 20;

@GET
public Cervejas
    listeTodasAsCervejas(@QueryParam("pagina") int pagina) {

    List<Cerveja> cervejas =
        estoque.listarCervejas(pagina, TAMANHO_PAGINA);

    return new Cervejas(cervejas);
}
```

Para testar, basta acessar a URL de antes. Primeiro, sem parâmetro algum:



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<cervejas>
  <link href="cervejas/Erdinger%20Weissbier" title="Erdinger Weissbier" rel="cerveja"/>
  <link href="cervejas/Stella%20Artois" title="Stella Artois" rel="cerveja"/>
</cervejas>
```

Figura 6.8: Primeiro teste de paginação

Depois, com o número da página igual a 0:



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<cervejas>
  <link href="cervejas/Erdinger%20Weissbier" title="Erdinger Weissbier" rel="cerveja"/>
  <link href="cervejas/Stella%20Artois" title="Stella Artois" rel="cerveja"/>
</cervejas>
```

Figura 6.9: Segundo teste de paginação

Agora, com o número da página igual a 1:



Figura 6.10: Terceiro teste de paginação

Note que agora não foram retornados resultados. Para que isto fique mais evidente, altere a constante `TAMANHO_PAGINA` no código para ter o valor 1 e refaça o teste:



Figura 6.11: Refazendo o teste



Figura 6.12: Refazendo o teste

6.9 ADAPTANDO O CÓDIGO PARA RETORNAR JSON

Até aqui, nós só estamos trabalhando com retorno de dados em XML. Para que seja possível trabalhar com JSON também,

precisamos fazer algumas alterações. A primeira delas é incluir as bibliotecas necessárias no *classpath* da aplicação. Por exemplo, se você usa Maven, basta incluir o seguinte no `pom.xml`:

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jettison</artifactId>
    <version>${jersey.version}</version>
</dependency>
```

Isso fará com que o Jettison e suas dependências sejam adicionadas ao *classpath* do sistema. O próximo passo é fazer com que o sistema do JAX-RS carregue a funcionalidade do JSON. Para isso, basta retornar à classe `ApplicationJAXRS` e sobrescrever o método `getSingletons` com o seguinte:

```
@Override
public Set<Object> getSingletons() {
    Set<Object> singletons = new HashSet<>();

    //org.glassfish.jersey.jettison.JettisonFeature
    singletons.add(new JettisonFeature());

    return singletons;
}
```

Finalmente, modifique a classe `CervejaService` para tratar do novo tipo de conteúdo:

```
@Path("/cervejas")
@Consumes({ MediaType.TEXT_XML, MediaType.APPLICATION_XML,
            MediaType.APPLICATION_JSON })
@Produces({ MediaType.TEXT_XML, MediaType.APPLICATION_XML,
            MediaType.APPLICATION_JSON })
public class CervejaService
```

6.10 IMPLEMENTANDO RETORNO DE DADOS BINÁRIOS

Para implementar o retorno da foto da cerveja, devemos implementar um método que atue sobre a mesma URL e sobre o

mesmo método, apenas alterando o `MIME Type` de retorno. Isto pode ser feito por meio da anotação `@Produces`, que vai sobrescrever a definição do que está na classe. Vamos utilizar a anotação `@Produces` usando o curinga descrito no capítulo *O protocolo HTTP*, isto é, `image/*`:

```
@GET  
@Path("{nome}")  
@Produces("image/*")
```

Outro desafio a ser resolvido é a questão do tipo de retorno. Queremos um método que responda a requisições de imagens, mas que não esteja preso a um tipo particular de imagem. Assim sendo, precisamos ajustar o `MIME Type` em tempo de execução – objetivo que pode ser alcançado a partir da classe `Response`. A assinatura do método fica assim:

```
@GET  
@Path("{nome}")  
@Produces("image/*")  
public Response  
    recuperaImagem(@PathParam("nome") String nomeDaCerveja)
```

Agora, é hora de implementar o método. Para isso, colocamos as imagens em uma pasta qualquer (no meu exemplo, coloquei na pasta `src/main/resources`, o que fará com que as fotos fiquem na raiz do código). Então, implementamos um método para leitura utilizando a API de I/O do Java:

```
// Estou utilizando .jpg no final porque as imagens ainda não  
// são dinâmicas  
InputStream is = CervejaService.class.getResourceAsStream("//"  
    + nomeDaCerveja + ".jpg");  
  
byte[] dados = new byte[is.available()];  
is.read(dados);  
is.close();
```

Com o array `dados`, podemos usar a classe `Response` para gerar o resultado. Lembre-se de que o retorno deste método tem o

código de status 200, e o tipo deve ser ajustado para `image/jpg` . Assim sendo, podemos utilizar o seguinte:

```
return Response.ok(dados).type("image/jpg").build();
```

O método completo, portanto, fica assim:

```
@GET  
@Path("{nome}")  
@Produces("image/*")  
public Response  
    recuperaImagem(@PathParam("nome") String nomeDaCerveja)  
    throws IOException {  
    InputStream is = CervejaService.class.getResourceAsStream("//"  
        + nomeDaCerveja + ".jpg");  
  
    if (is == null)  
        throw new WebApplicationException(Status.NOT_FOUND);  
  
    byte[] dados = new byte[is.available()];  
    is.read(dados);  
    is.close();  
  
    return Response.ok(dados).type("image/jpg").build();  
}
```

Para testar esse código, basta usar o `curl` . Por exemplo, se quiser salvar o retorno deste serviço em um arquivo (no Linux), utilize o seguinte comando:

```
curl --url http://localhost:8080/cervejaria/cervejas/  
Stella%20Artois -H "Accept: image/*" >> imagem.jpg
```

A seguir, abra o arquivo `imagem.jpg` .

6.11 IMPLEMENTANDO CONSUMO DE DADOS BINÁRIOS

Para realizar o processo inverso, devemos modelar um método que receba uma imagem pelo método `POST` . Vamos começar modelando o método, então, para que responda a este e também receba o nome da cerveja por meio da URL:

```
@POST  
@Path("{nome}")  
public Response  
    criaImagem(@PathParam("nome")String nomeDaImagem)
```

O próximo passo é fazer esse método ser capaz de receber qualquer imagem. Lembre-se de que podemos fazer isso pelo uso do curinga, ou seja, o `*` em `image/*`. Assim sendo, anotamos o método com `@Consumes("image/*")`:

```
@POST  
@Path("{nome}")  
@Consumes("image/*")  
public Response  
    criaImagem(@PathParam("nome")String nomeDaImagem)
```

Para sermos capazes de detectar o tipo da imagem fornecido pelo cliente, recorremos ao uso da própria API de servlets do Java. Podemos utilizar a interface `HttpServletRequest` para recuperar o tipo de dados (assim como visto no capítulo anterior). Antes, precisamos injetar esta interface no nosso método via anotação `javax.ws.rs.core.Context`:

```
@POST  
@Path("{nome}")  
@Consumes("image/*")  
public Response  
    criaImagem(@PathParam("nome")String nomeDaImagem,  
               @Context HttpServletRequest req)
```

Finalmente, precisamos acessar os dados propriamente ditos. Fazer isto é simples: basta inserir na assinatura do método um array de bytes, que vai conter a imagem:

```
@POST  
@Path("{nome}")  
@Consumes("image/*")  
public Response  
    criaImagem(@PathParam("nome")String nomeDaImagem,  
               @Context HttpServletRequest req,  
               byte[] dados)
```

Agora, vamos à implementação. Como ainda não temos um

banco de dados no nosso sistema, ou algo assim, façamos pura e simplesmente a gravação da nossa imagem no sistema de arquivos (na *home* do usuário. No Windows, por exemplo, esse diretório é C:\Users\<nome do usuário> ; no Linux, é /home/<nome do usuário>). O código é o seguinte:

```
private static Map<String, String> EXTENSOES;

static {
    EXTENSOES = new HashMap<>();
    EXTENSOES.put("image/jpg", ".jpg");
}

@POST
@Path("{nome}")
@Consumes("image/*")
public Response
    criaImagem(@PathParam("nome")String nomeDaImagem,
               @Context HttpServletRequest req,
               byte[] dados) throws IOException, InterruptedException {

    String userHome = System.getProperty("user.home");
    String mimeType = req.getContentType();
    FileOutputStream fos = new FileOutputStream(userHome
        + java.io.File.separator + nomeDaImagem
        + EXTENSOES.get(mimeType));

    fos.write(dados);
    fos.flush();
    fos.close();

    return Response.ok().build();
}
```

Para testar, tenha um arquivo .jpg contendo uma cerveja. Por exemplo, no código-fonte deste livro, está disponível uma imagem chamada Stella Artois.jpg . Assim sendo, para realizar o upload desta imagem, basta executar o seguinte comando do curl :

```
curl -H "Content-Type: image/jpg" --data-binary
"@Stella      Artois.jpg"          -X POST      -v
http://localhost:8080/cervejaria/cervejas/Stella%20Artoi:
. Uma vez finalizado, cheque o diretório do seu usuário para
```

verificar se existe um arquivo chamado `Stella Artois.jpg` criado lá.

6.12 CONCLUSÃO

Neste capítulo, você viu como utilizar a API do Java para serviços REST, a JAX-RS, e como esta API economiza o trabalho que tivemos no capítulo anterior, automatizando grande parte do esforço que, de outra forma, seria repetitivo e moroso.

Muita coisa ainda não foi explicada, no entanto. E quanto ao código dos clientes? E quanto à segurança? E como resolver problemas mais complexos?

Nos próximos capítulos, você deverá visualizar as respostas para estas e várias outras questões.

REST, CLIENT-SIDE

"Uma vida sem desafios não vale a pena ser vivida." – Sócrates

Neste ponto, nosso serviço de manutenção de cervejas está construído e em pleno funcionamento. Agora, precisamos definir as maneiras pela qual os nossos clientes poderão utilizar este serviço.

Neste capítulo, veremos como construir clientes, como verificar contratos de funcionamento dos serviços e como desenvolver testes integrados.

7.1 JAVASCRIPT

Caso a nossa cervejaria possua uma interface online, é interessante que se possa consumir nossos serviços REST diretamente a partir do browser, trazendo a vantagem de não ser necessário construir código extra no *server-side* para consumir estes serviços. Isso pode ser feito a partir da linguagem **JavaScript**, que é executada diretamente no browser.

Existe uma grande inconveniência na construção de clientes JavaScript para serviços REST por meio de browsers: a implementação da maneira de consumir estes serviços varia entre os diversos browsers existentes (`Chrome` , `Firefox` , `Internet Explorer` etc.). Para contornarmos este problema, faremos uso do framework `jQuery`, bastante popular neste tipo de cliente.

Nossa página inicial (`index.html`) vai começar, então, com a seguinte estrutura:

```
<html>
  <head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8">
    <title>Cervejaria REST</title>
    <script src="jquery-1.10.2.min.js"></script>
  </head>
  <body>

    <fieldset>
      <legend>Listagem de cervejas</legend>
      <table id="grid">
        <thead>
          <tr>
            <th>Nome</th>
            <th>Cervejaria</th>
            <th>Descrição</th>
            <th>Tipo</th>
            <th>Opções</th>
          </tr>
        </thead>
        <tr>
          <td colspan="5">Carregando...</td>
        </tr>
      </table>
    </fieldset>
  </body>
</html>
```

Note que também é necessário incluir o jQuery no contexto da nossa página. Basta copiar e colar o conteúdo de <http://code.jquery.com/jquery-1.10.2.min.js> no arquivo `jquery-1.10.2.min.js` (ou então colocar este endereço diretamente na importação descrita pela tag `script`). A estrutura de pastas ficará assim:

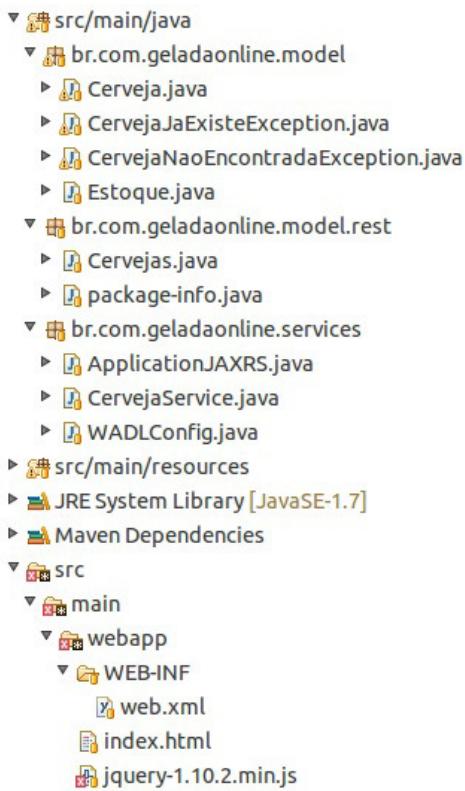


Figura 7.1: Estrutura física do projeto

No entanto, se você inicializar o sistema e acessar o endereço <http://localhost:8080/cervejaria/index.html> a partir do seu browser, deverá receber o seguinte erro:

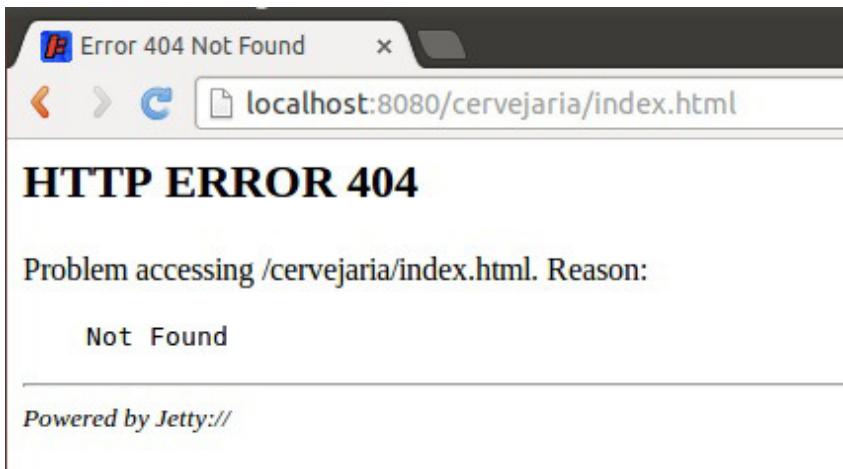


Figura 7.2: Erro

Por que isso aconteceu? O *servlet* do Jersey tem prioridade no sistema de mapeamento do Jetty, logo, ele procurou por um serviço disponível em `index.html` e, obviamente, não encontrou — daí o retorno do código 404. Primeiro precisamos, então, resolver esse conflito.

Para isso, basta mudar a URL de disponibilização do *servlet*. Acesse o arquivo `web.xml` e mude a seguinte estrutura:

```
<servlet-mapping>
    <servlet-name>
        br.com.geladaonline.services.ApplicationJAXRS
    </servlet-name>
    <url-pattern>/services/*</url-pattern>
</servletmapping>
```

Agora, ao acessar novamente a página, vemos o seguinte:



Figura 7.3: Acessando a página novamente

Agora sim, podemos prosseguir com a criação do nosso cliente JavaScript. Já que modificamos a URL dos nossos serviços (e isso deve ser constante entre modificações de ambientes desenvolvimento / homologação / produção), parece prudente criar uma variável para armazenar o endereço atual dos serviços. Assim sendo, criaremos o seguinte:

```
<script type="text/javascript">
host = "http://localhost:8080/cervejaria/services/";
</script>
```

Na sequência, crie uma função `listarCervejas`, que deverá ser invocada uma vez que a carga da página for concluída. Esta função será responsável por carregar as cervejas a partir do serviço de listagem. Lembre-se de que esse serviço retorna links para as outras cervejas, logo, essa função deverá ser responsável por seguir estes links e atualizar a página conforme necessário.

No corpo dessa função, você usará o módulo AJAX do jQuery. Este módulo é responsável por realizar requisições HTTP externas, e tem o seguinte formato geral:

```
$.ajax({
  url : 'http://exemplo.com/recurso',
  type : 'GET,POST,PUT,DELETE,etc.',
  success : function(data) {},
```

```
        error: function(data)
    })
}
```

Ou seja, para realizar uma chamada REST a partir desse componente, basta utilizar o \$ (que é um *alias* para o jQuery) e invocar a função ajax , passando como parâmetros os dados aqui exemplificados. Por exemplo, no nosso caso, para realizar uma chamada para o serviço de listagem de cervejas, usamos os seguintes parâmetros:

```
function listarCervejas() {
    $.ajax({
        url : host + 'cervejas',
        type : 'GET',
        success : function(data) {
            alert("Dados retornados com sucesso");
        },
        error: function(data) {
            alert("Erro na invocação");
        }
    });
}
```

Ao pé do elemento body , inclua o seguinte código (para disparar a execução da função listarCervejas):

```
<script type="text/javascript">
    listarCervejas();
</script>
```

Agora, ao recarregar a página, você deve ver algo semelhante ao seguinte:

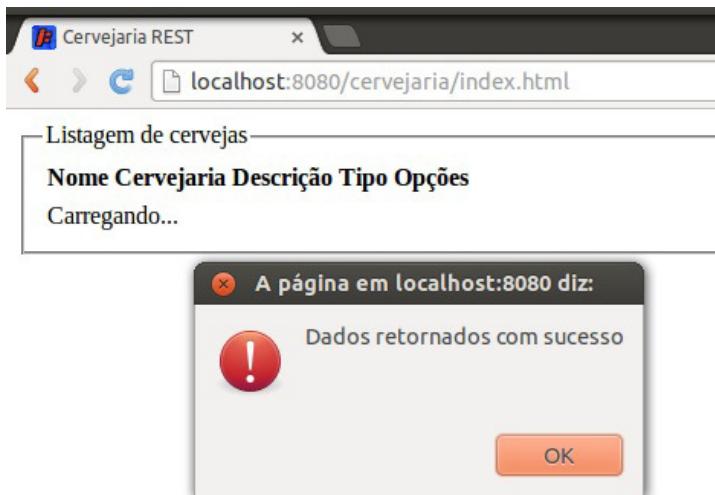


Figura 7.4: Primeira execução

A exibição desta caixa de mensagem indica que a função foi executada e os dados foram retornados com sucesso, provocando a execução da função mapeada em `success`.

ESTA CAIXA NÃO APARECEU, O QUE FAÇO?

Se nenhuma mensagem tiver aparecido, significa que há um erro na sintaxe do código. Se a caixa tiver aparecido com a mensagem "Erro na invocação", significa que o código está correto, mas houve um problema na invocação do serviço.

No último caso, sugiro conferir a URL usada para recuperar as cervejas. Pode haver um erro de digitação no conteúdo da variável `host` — note que ela é encerrada com uma barra. No primeiro, sugiro usar um *debugger* do browser para verificação de erros. Por exemplo, se você utiliza o Google Chrome, basta apertar `F12` para ter acesso ao seu debugger.

O próximo passo agora é incrementar a função de sucesso para que ela siga os links das cervejas. Para isso, é necessário ter em mente os seguintes passos:

- Para cada execução deste método, será necessário limpar a tabela para, então, repopulá-la.
- Caso um único link seja retornado, o jQuery considera-o como uma única propriedade de cerveja ; caso contrário, ele considera link como um *array*. Ou seja, o tratamento será diferenciado em cada caso.

Para realizar a limpeza da tabela, é simples. O jQuery trabalha fortemente com o conceito de **seletores**, que são *strings* que facilitam a localização de determinado(s) elemento(s). Por exemplo, para localizar a tabela (cujo *id* é `grid`), basta utilizar uma *string* começando com `#` , seguido desse *id*, assim:

```
$( '#grid' )
```

Desta forma, o jQuery vai selecionar a tabela como um todo. Para selecionar apenas as suas linhas, usamos um subseletor para localizar os elementos `tr` (que são as tags que delimitam cada linha da tabela). Por fim, aplicamos uma restrição na seleção de linhas: o jQuery deverá selecionar apenas a segunda linha e subsequentes, posto que já existe uma linha no cabeçalho da tabela. O código para seleção fica assim:

```
$( '#grid tr:gt(0)' )
```

Finalmente, utilizamos o método `remove()` , que será responsável por arrancar os elementos selecionados da tabela:

```
$( '#grid tr:gt(0)' ).remove();
```

O próximo passo é detectar se `link` é um *array* ou uma propriedade. Isso pode ser detectado pela função `isArray` do

jQuery:

```
if ($.isArray(data.cervejas.link)) {  
    for ( var i = 0; i < data.cervejas.link.length; i++) {  
        var link = data.cervejas.link[i]['@href'];  
    }  
} else {  
    var link = data.cervejas.link['@href'];  
}
```

Assim sendo, a variável `link` vai conter o conteúdo do atributo `href` em ambos os casos.

Finalmente, vamos criar uma função que vai receber este atributo e realizar uma nova invocação para poder preencher a tabela. A nova função vai ser bastante semelhante à primeira:

```
function segueLinkCerveja(link) {  
    $.ajax({  
        url : host + link,  
        type : 'GET',  
        success : function(data) {  
            adicionaCervejaNovaAoGrid(data.cerveja);  
        },  
        error : function(data) {  
            alert("Ocorreu um erro");  
        }  
    });  
}
```

Note que aqui a diferença primordial está no fato de que a função `adicionaCervejaNovaAoGrid` ainda não existe. Logo, precisamos criá-la. Esta função terá tão somente o objetivo de criar o HTML que deverá ser incluído na tabela, ou seja, deverá apresentar os atributos de cada cerveja encontrada pelo serviço.

Assim sendo, a função terá o seguinte código:

```
function adicionaCervejaNovaAoGrid(cerveja) {  
  
    var data = "<tr>"  
    + "<td>" + cerveja.nome + "</td>"  
    + "<td>" + cerveja.cervejaria + "</td>"  
    + "<td>" + cerveja.descricao + "</td>"
```

```

        + "<td>" + cerveja.tipo + "</td>"  

        + "<td><input type=\"button\" value=\"Apagar\" /></td>"  

        + "</tr>";  
  

    $("#grid").append(data);  

}

```

Finalmente, alteramos novamente o código que recupera os links e os passamos como parâmetro para a função segueLinkCerveja :

```

if ($.isArray(data.cervejas.link)) {  

    for ( var i = 0; i < data.cervejas.link.length; i++) {  

        var link = data.cervejas.link[i]['@href'];  

        segueLinkCerveja(link);  

    }  

} else {  

    var link = data.cervejas.link['@href'];  

    segueLinkCerveja(link);  

}

```

Ao entrar na página, visualizamos o seguinte:

Listagem de cervejas					
Nome	Cervejaria	Descrição	Tipo	Opções	
Stella Artois	Artois	A cerveja belga mais francesa do mundo :)	LAGER	Apagar	
Erdinger Weissbier	Erdinger Weissbräu	Cerveja de trigo alemã	WEIZEN	Apagar	

Figura 7.5: Busca completa

Criando novas cervejas

A busca pelas cervejas por meio do cliente JavaScript está concluída. Agora, podemos incrementar o cliente realizando a criação de uma cerveja. Para isso, precisamos criar um formulário, para que o cliente possa submeter os dados de uma cerveja a ser criada. O código do formulário pode ser o seguinte:

```
<fieldset>
```

```

<legend>Criar nova cerveja</legend>
<form id="criarCervejaForm">
    <label>Nome</label>
    <input type="text" name="nome" />
    <br />
    <label>Cervejaria</label>
    <input type="text" name="cervejaria" />
    <br />
    <label>Descrição</label>
    <input type="text" name="descricao" />
    <br />
    <label>Tipo</label>
    <select name="tipo">
        <option value="LAGER" selected="selected">
            Lager
        </option>
        <option value="PILSEN">Pilsen</option>
        <option value="BOCK">Bock</option>
        <option value="WEIZEN">Weizen</option>
    </select>
    <input type="button" value="Criar"
        onclick="adicionaCerveja();" />
</form>
</fieldset>

```

Observe que cada um dos elementos `input` possui um atributo `name` correspondente a atributos da cerveja. Isto não é coincidência; vamos utilizar este formulário da forma como está para enviarmos como requisição ao nosso serviço.

A exceção à regra é o último `input`, que é o botão que vai disparar a função JavaScript `adicionaCerveja`. Ou seja, precisamos criar esta função. Observe que a lógica incluída nela seguirá a mesma estrutura das requisições `GET`, mas alterada para realizar um `POST`:

```

function adicionaCerveja() {
    $.ajax({
        url : host + 'cervejas',
        type : 'POST',
        contentType : 'application/json',
        data : data,
        success : function(data) {
            alert("Incluído com sucesso!");

```

```

        listarCervejas();
    },
    error : function(data) {
        alert("Ocorreu um erro");
    }
});
}

```

Se você observar com atenção, perceberá que dois valores foram adicionados aos parâmetros: `contentType` e `data`. Estes são responsáveis por dizer ao serviço, respectivamente, qual o tipo de conteúdo (pelo cabeçalho `Content-Type`) e os dados que serão fornecidos ao serviço, pelo corpo da requisição. No entanto, estes dados ainda não foram criados.

Para criá-los, vamos serializar o formulário em formato JSON. Isso pode ser feito por meio da adição de um *plugin* jQuery. No meu caso, obtive um desenvolvido por Mario Izquierdo, disponível em <https://github.com/marioizquierdo/jquery.serializeJSON>. Copiei o conteúdo do plugin e coleei no arquivo `json.js` — assim, basta realizar a inclusão do plugin com a diretiva:

```
<script src="json.js"></script>
```

A seguir, realizamos a criação do conteúdo a ser fornecido para o serviço usando os métodos `serializeJSON` e `JSON.stringify`, assim:

```
var data = $("#criarCervejaForm").serializeJSON();
data = "{\"cerveja\"::" + JSON.stringify(data) + "}";
```

A função completa fica:

```
function adicionaCerveja() {

    var data = $("#criarCervejaForm").serializeJSON();
    data = "{\"cerveja\"::" + JSON.stringify(data) + "}";

    $.ajax({
        url : host + 'cervejas',
        type : 'POST',
        contentType : 'application/json',
```

```

        data : data,
        success : function(data) {
            alert("Incluído com sucesso!");
            listarCervejas();
        },
        error : function(data) {
            alert("Ocorreu um erro");
        }
    });
}

```

Ao executarmos a criação de uma nova cerveja, obtemos o seguinte:

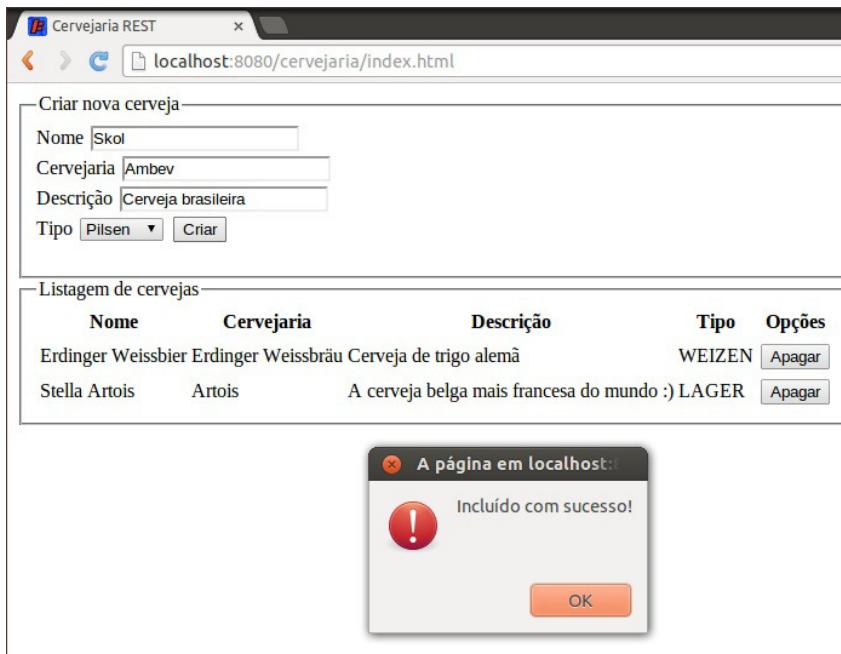


Figura 7.6: Incluindo a cerveja

Figura 7.7: Sucesso!

Observe que, caso você tente criar novamente uma cerveja com o mesmo nome, um erro ocorrerá. Isso é porque o serviço não é idempotente e não aceita cervejas com o mesmo nome. Assim sendo, é importante limpar o formulário de criação uma vez que os dados sejam incluídos. Isso pode ser feito com o seguinte código:

```
$("#criarCervejaForm")[0].reset();
```

Apagando uma cerveja criada

Finalmente, vamos criar código que faz com que uma dada cerveja seja excluída do serviço. Para isso, basta incluir novamente a chamada ajax do jQuery, utilizando o método `DELETE` :

```
function apagaCerveja(id) {
    $.ajax({
        url : host + 'cervejas/' + id,
        type : 'DELETE',
        success : function(data) {
            listarCervejas();
        },
        error : function(data) {
            alert("Ocorreu um erro");
        }
    });
}
```

```
}
```

Agora, precisamos criar o código que chama este método. Para isso, precisamos apenas alterar o código de geração do botão Apagar , passando como parâmetro o nome de cada cerveja:

```
function adicionaCervejaNovaAoGrid(cerveja) {  
  
    var data = "<tr>"  
    + "<td>" + cerveja.nome + "</td>"  
    + "<td>" + cerveja.cervejaria + "</td>"  
    + "<td>" + cerveja.descricao + "</td>"  
    + "<td>" + cerveja.tipo + "</td>"  
    + "<td><input type=\"button\" value=\"Apagar\" \"  
    + "onclick=\"apagaCerveja('" + cerveja.nome + "');\" />  
    + "</td>"  
    + "</tr>";  
  
    $("#grid").append(data);  
}
```

Quando clicarmos no botão Apagar de qualquer uma das cervejas, a função apagaCerveja será chamada, passando como parâmetro o nome da cerveja a ser apagada.

Tratamento de erros

O último retoque a ser dado ao código é uma maneira de avisar ao cliente sobre erros. Por exemplo, no caso da criação da cerveja, qual mensagem o cliente deverá visualizar no caso de tentativa de criação de uma cerveja que já existe?

Deixo por sua conta, leitor, um tratamento mais rebuscado. Aqui, vou apenas abordar a exibição do código de status e do texto descritivo. Para obtê-los, basta acessar as propriedades status e statusText , respectivamente. Assim, temos o seguinte código de erros:

```
error : function(data) {  
    alert("Ocorreu um erro: " + data.status + " " +  
        data.statusText);
```

}

Quando tentamos criar uma cerveja com um nome que já existe, obtemos o seguinte:

The screenshot shows a browser window titled 'Cervejaria REST' with the URL 'localhost:8080/cervejaria/index.html'. The page has two sections: 'Criar nova cerveja' and 'Listagem de cervejas'. In the 'Criar nova cerveja' section, fields for Nome (Stella Artois), Cervejaria (Artois), Descrição (A cerveja belga mais francesa do mundo :)), and Tipo (Lager) are filled out, with a 'Criar' button. Below this is a table titled 'Listagem de cervejas' with columns: Nome, Cervejaria, Descrição, Tipo, and Opções. It lists two rows: Erdinger Weissbier (Erdinger Weissbräu, Cerveja de trigo alemã, WEIZEN, Apagar) and Stella Artoois (Artois, A cerveja belga mais francesa do mundo :), LAGER, Apagar). A modal dialog box titled 'A página em localhost:8080 diz:' displays an error message: 'Ocorreu um erro: 409 Conflict' with an exclamation mark icon, and an 'OK' button.

Figura 7.8: Conflito

7.2 JAX-RS

A própria especificação JAX-RS, a partir da versão 2.0, oferece uma API para criação de clientes REST em Java. Ela foi fortemente baseada na API do Jersey (ainda na versão 1.x), e tem por conceito ser independente da implementação *server-side*, ou seja, pode consumir serviços REST criados em qualquer linguagem.

Para criar este cliente, vamos fazer outro projeto que contenha as classes necessárias. Este outro projeto deverá conter os JARs da API do JAX-RS. Se você usa Maven, basta utilizar o seguinte:

```
<dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>${jersey.version}</version>
</dependency>
```

A seguir, vamos começar a criar o cliente em si. Todo código de clientes JAX-RS gira em torno da interface `javax.ws.rs.client.Client`, que é instanciada a partir da classe `javax.ws.rs.client.ClientBuilder`:

```
import javax.ws.rs.client.*;
public class Cliente {
    public static void main(String[] args) {
        Client client = ClientBuilder.newClient();
    }
}
```

A interface `Client` oferece uma API fluente para que seja possível detalhar cada requisição. Ao final da preparação da requisição, pode-se submetê-la e recuperar uma classe Java de modelo já populada. No caso, as classes de modelo são `Cervejas` e `Cerveja`, e basta copiá-las do projeto do servidor para o projeto do cliente. Até o momento, a estrutura do projeto deve estar assim:

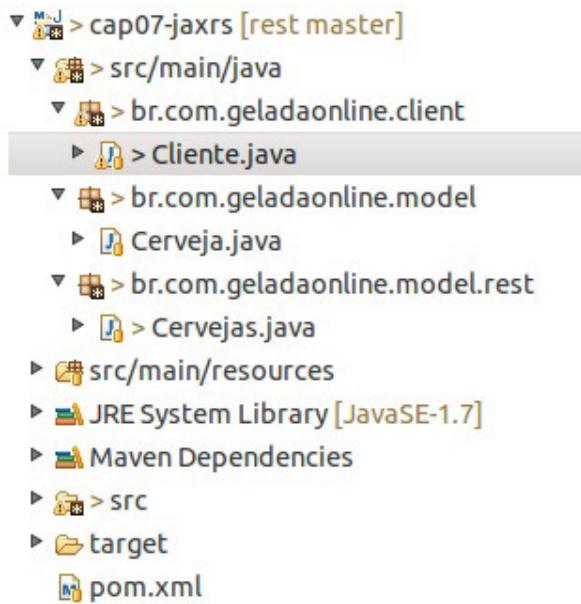


Figura 7.9: Estrutura do projeto

O próximo passo é realizar adaptações para que seja possível interagir com os links HATEOAS. Infelizmente, até o momento em que escrevo este livro, o suporte a estes links por parte da API é limitado, contendo um bug no momento do mapeamento dos links que estão contidos no corpo da resposta para o objeto Java. Logo, é necessário realizar um *workaround* para contornar essa limitação, por meio da criação de uma classe que pode ser mapeada para o Java:

```
package br.com.geladaonline.model.rest;

import javax.xml.bind.annotation.*;

@XmlAccessorType(XmlAccessType.FIELD)
public class CustomLink {

    @XmlAttribute
    private String href;

    @XmlAttribute
```

```

private String rel;

@XmlAttribute
private String title;

// getters e setters
}

```

A partir dessa classe, utilizamos a seguinte estratégia: o método `getLinks` da classe `Cervejas` será mantido com a mesma assinatura que possui do lado do servidor. No entanto, a implementação será modificada, de maneira que o JAXB injete na classe instâncias de `CustomLink` (via injeção direta de atributos). Quando nosso código for acessar o método `getLinks`, instâncias de `javax.ws.rs.core.Link` serão fornecidas. Ou seja, as instâncias de `CustomLink` (a classe que acabamos de criar) serão convertidas em links da API do JAX-RS.

Dessa forma, modifica-se o conteúdo da classe `Cervejas` para ficar assim:

```

import javax.ws.rs.core.Link;
import javax.xml.bind.annotation.*;

@XmlElement(name = "link")
private List<CustomLink> links;

public List<Link> getLinks() {
    return null;
}
}

```

Vamos criar uma interface para armazenar o endereço dos serviços:

```

package br.com.geladaonline.client;

public interface Constants {

```

```
    public static final String HOST =
        "http://localhost:8080/cervejaria/services/";

}
```

Agora, a implementação do método `getLinks` será bastante similar à implementação do lado do servidor:

```
public List<Link> getLinks() {
    List<Link> links = new ArrayList<>();

    for (CustomLink customLink : this.links) {

        Link newLink =
            // É necessário inserir o host, pois os links retornados pelo
            // servidor são relativos, i.e., não têm essa informação
            Link.fromUri(Constants.HOST + customLink.getHref())
                .rel(customLink.getRel())
                .title(customLink.getTitle())
                .build();

        links.add(newLink);
    }

    return links;
}
```

Agora, ao retomar o código da classe `Cliente`, temos condições de fornecer todos os dados para a invocação do serviço. Você já sabe que as três informações fundamentais para os serviços REST são o endereço (URL), o método HTTP e o tipo de conteúdo. Assim sendo, essas são as informações que temos de fornecer.

Comecemos pela URL: sua composição é feita a partir dos métodos `target` e `path`. Esta divisão é feita para facilitar a substituição entre ambientes (desenvolvimento / homologação / produção, por exemplo), já que o método `target` recebe como parâmetro a URL de base — ou seja, que será modificada conforme as migrações do código de ambiente para ambiente — e o método `path` recebe o caminho do recurso solicitado, propriamente dito.

Para que possamos fazer uma requisição para o serviço de cervejas, preparamos a URL da seguinte maneira:

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

WebTarget target = client.target(Constants.HOST);
target.path("cervejas");
```

Ou, seguindo a interface fluente:

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

client.target(Constants.HOST).path("cervejas");
```

O próximo passo é ajustar o MIME Type da requisição. Isto é feito a partir do método `request`:

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

client
    .target(Constants.HOST)
    .path("cervejas")
    .request("application/xml");
```

Note que o método `request` é sobrecarregado. Ele aceita tanto uma `String` como parâmetro quanto os valores enumerados na classe `javax.ws.rs.core.MediaType`. Também pode ser utilizado sem parâmetros; nesse caso, o Jersey fornecerá um valor padrão, que é `text/html`, `image/gif`, `image/jpeg`, `*; q=.2, */*; q=.2`. Ou seja, ele solicita preferencialmente `text/html`, `image/gif` ou `image/jpeg`. Caso nenhum destes esteja disponível, deixa a cargo do servidor decidir o melhor tipo disponível.

O último passo é ajustar o método HTTP. Esta parte é trivial: o nome dos métodos HTTP dão os nomes dos métodos da interface fluente. Assim, se queremos usar o método HTTP `GET`, basta utilizar o método `get()` no Java. Este método também é sobrecarregado, e podemos passar como parâmetro a classe `Cervejas` para indicar que queremos que a resposta seja dada como uma instância de `Cervejas`:

```
Cervejas cervejas = client
    .target(Constants.HOST)
    .path("cervejas")
    .request("application/xml")
    .get(Cervejas.class);
```

Ao executar este código, a requisição será feita, e teremos o resultado na forma especificada pela classe `Cervejas`. Ou seja, se executarmos:

```
System.out.println(cervejas);
System.out.println(cervejas.getLinks().size());
```

Obteremos algo semelhante ao seguinte:

```
br.com.geladaonline.model.rest.Cervejas@2c98df31
2
```

Note que o método `getLinks` retorna uma lista com tipo genérico `javax.ws.rs.core.Link`. Esta classe é bastante útil para que possamos seguir os links; basta usar o método `invocation()` que, por sua vez, substituirá os métodos `target()` e `path()`:

```
List<Cerveja> cervejaList = new ArrayList<>();

for (Link link : cervejas.getLinks()) {
    Cerveja cerveja = ClientBuilder.newBuilder()
        .invocation(link)
        .accept(MediaType.APPLICATION_XML)
        .get(Cerveja.class);

    cervejaList.add(cerveja);
}
```

Além disso, também podemos substituir o tipo da requisição, e solicitar JSON, por exemplo. Nesse caso, é necessário informar ao Jersey qual o conversor. Como estamos utilizando Jettison do lado do servidor, é altamente recomendado usarmos Jettison também do lado do cliente (para evitar problemas como os detalhados no capítulo *Tipos de dados*). Basta adicionar a seguinte dependência no Maven:

```
<dependency>
```

```
<groupId>org.glassfish.jersey.media</groupId>
<artifactId>jersey-media-json-jettison</artifactId>
<version>${jersey.version}</version>
</dependency>
```

Então, registramos o conversor a partir do método `register`:

```
List<Cerveja> cervejaList = new ArrayList<>();

for (Link link : cervejas.getLinks()) {
    Cerveja cerveja = ClientBuilder.newBuilder()
        .register(org.glassfish.jersey.jettison
            .JettisonFeature.class)
        .invocation(link)
        .accept(MediaType.APPLICATION_JSON)
        .get(Cerveja.class);

    cervejaList.add(cerveja);
}
```

COMO TIRAR A LIMPO O QUE ESTÁ SENDO EFETIVAMENTE RETORNADO?

Caso você tenha a curiosidade (ou desconfiança) de saber como os dados estão sendo efetivamente retornados, basta utilizar o conversor da própria API. Ou seja, em vez de realizar a conversão para `Cervejas`, faça para `String`.

Criando uma nova cerveja

Para criar uma nova cerveja usando a API do JAX-RS, o processo é similar. O que muda é que o método `post()`, em vez de aceitar apenas um parâmetro, aceita dois: a entidade a ser enviada e o modelo do retorno (este, semelhante ao retorno do método `get()`).

Para que seja possível fornecer esse parâmetro, a API fornece a classe `javax.ws.rs.client.Entity`, que será responsável por fornecer o MIME Type da entidade (entre outros detalhes). Esta

classe possui o método `entity` , que aceita como parâmetro o objeto Java com os dados e o `MIME Type` :

```
private static Cerveja criarCerveja(Cerveja cerveja) {  
    cerveja = ClientBuilder.newClient()  
        .target(Constants.HOST)  
        .path("cervejas")  
        .request()  
        .post(Entity.entity(cerveja, MediaType.APPLICATION_XML),  
              Cerveja.class);  
  
    return cerveja;  
}
```

Quando está realizando o envio de um tipo `application/xml` , o método `entity` pode ser substituído simplesmente por `xml()` :

```
private static Cerveja criarCerveja(Cerveja cerveja) {  
    cerveja = ClientBuilder.newClient()  
        .target(Constants.HOST)  
        .path("cervejas")  
        .request()  
        .post(Entity.xml(cerveja), Cerveja.class);  
  
    return cerveja;  
}
```

Além disso, a API também permite que não façamos a conversão para `Cerveja` . Você deve se lembrar de que a criação da cerveja retorna o código de status `201 Created` , bem como o cabeçalho `Location` com a localização do recurso. Vamos exercitar a localização deste recurso.

Para isso, em vez de fornecer o parâmetro de conversão da resposta, vamos deixar de passar este parâmetro, provocando o retorno de uma `javax.ws.rs.core.Response` :

```
Response response = ClientBuilder  
    .newClient()  
    .target(Constants.HOST)  
    .path("cervejas")  
    .request()  
    .post(Entity.xml(cerveja));
```

Uma vez retornada a `Response`, vamos recuperar o conteúdo do cabeçalho `Location` e convertê-lo em um `Link` por meio do método `getLocation`:

```
Link link = Link.fromUri(response.getLocation()).build();  
  
cerveja = ClientBuilder  
    .newClient()  
    .invocation(link)  
    .accept(MediaType.APPLICATION_XML)  
    .get(Cerveja.class);
```

A classe `Response` também pode ser usada para fazer comparações a respeito do código de retorno:

```
if (response.getStatus() == Response.Status.CREATED  
        .getStatusCode()) {  
    //recupera o cabeçalho Location  
}
```

7.3 IMPLEMENTAÇÃO DOS CLIENTES COM O RESTEASY

Um framework bastante popular para construção de serviços é o RESTEasy, desenvolvido pela divisão JBoss da Red Hat (sendo um de seus principais desenvolvedores Bill Burke, um conhecido autor e arquiteto Java). No que tange a frameworks REST, o RESTEasy se destaca por oferecer um sistema *client-side* mais amigável a desenvolvedores Java, através do oferecimento de comunicação pela utilização de interfaces criadas pelo próprio usuário.

Vejamos, então, como o RESTEasy funciona. Para que comecemos, basta criar um novo projeto e adicionar as dependências do RESTEasy a ele. Se você utiliza Maven, use a seguinte declaração de dependências:

```
<properties>  
    <resteasy.version>3.0.6.Final</resteasy.version>  
</properties>
```

```

<dependencies>
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-client</artifactId>
        <version>${resteasy.version}</version>
    </dependency>
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-jettison-provider</artifactId>
        <version>${resteasy.version}</version>
    </dependency>
</dependencies>

```

Agora, vamos à criação da interface do nosso serviço. Basicamente, o que vamos fazer será colocar as assinaturas dos métodos do nosso serviço REST em uma interface que ficará no *client-side*:

```

package br.com.geladaonline.client;

//imports omitidos

@Path("/cervejas")
@Consumes({ MediaType.TEXT_XML, MediaType.APPLICATION_XML,
    MediaType.APPLICATION_JSON })
@Produces({ MediaType.TEXT_XML, MediaType.APPLICATION_XML,
    MediaType.APPLICATION_JSON })
public interface CervejaService {

    @GET
    @Path("{nome}")
    public Cerveja encontreCerveja(
        @PathParam("nome") String nomeDaCerveja);

    @GET
    public Cervejas listeTodasAsCervejas(
        @QueryParam("pagina") int pagina);

    @POST
    public Response criarCerveja(Cerveja cerveja);

    @PUT
    @Path("{nome}")
    public void atualizarCerveja(
        @PathParam("nome") String nome, Cerveja cerveja);

    @DELETE

```

```

@Path("{nome}")
public void apagarCerveja(@PathParam("nome") String nome) ;

@GET
@Path("{nome}")
@Produces("image/*")
public Response recuperarImagem(
    @PathParam("nome") String nomeDaCerveja);

@POST
@Path("{nome}")
@Consumes("image/*")
public Response criaImagem(@PathParam("nome")
    String nomeDaImagem, @Context HttpServletRequest req,
    byte[] dados);

}

```

Quanto às classes de modelo, utilize as mesmas criadas para o cliente JAX-RS puro (incluindo `CustomLink`).

A criação dos clientes RESTEasy neste formato é feita a partir da classe `org.jboss.resteasy.client.jaxrs.ProxyBuilder` . A API por trás dela faz com que uma classe seja criada, em tempo de execução, que implemente uma interface (no nosso caso, `CervejaService`). A classe criada fará as requisições no formato especificado pela interface.

A classe `ProxyBuilder` oferece dois métodos estáticos: `builder` e `proxy` . O método `builder` recebe como parâmetros a interface sobre a qual queremos criar um `proxy` e também uma instância de `WebTarget` — a mesma usada pelos clientes JAX-RS puros. O retorno deste método é uma instância da própria classe `ProxyBuilder` . A partir desta instância, é possível configurar algumas propriedades ou construir diretamente o `proxy`, ao utilizar o método `build` . Este método retorna a classe `proxy` — que atribuímos, então, a uma variável tipada pela interface:

```

package br.com.geladaonline.client;

import javax.ws.rs.client.ClientBuilder;

```

```

import org.jboss.resteasy.client.jaxrs.ProxyBuilder;
import br.com.geladaonline.model.rest.Cervejas;
public class Cliente {
    public static void main(String[] args) throws JAXBException {
        ProxyBuilder<CervejaService> proxy = ProxyBuilder.builder(
            CervejaService.class,
            ClientBuilder.newClient().target(
                "http://localhost:8080/cervejaria/services"));
        CervejaService service = proxy.build();
    }
}

```

A partir dessa variável, podemos interagir normalmente com o serviço. Por exemplo, se fizermos a solicitação de todas as cervejas (começando pela página zero), obteremos os links das cervejas normalmente:

```

Cervejas cervejas = service.listeTodasAsCervejas(0);
System.out.println(cervejas.getLinks().get(0));

```

Esse código faz com que o seguinte seja impresso no console:

```

<http://localhost:8080/cervejaria/cervejas/Erdinger%20Weissbier>;
title="Erdinger Weissbier"; rel="cerveja"

```

7.4 WADL

Um leitor atento provavelmente está se perguntando a essa altura: como os clientes dos meus serviços saberão qual formato de dados deve ser enviado para o serviço? Quer dizer, até o momento, temos um serviço de cervejas que, portanto, recebe e devolve cervejas em vários formatos. Mas o que uma cerveja deve conter? Como os meus clientes vão saber que a tag de nome é chamada `nome`, e não `Nome`, por exemplo? Existe algo que indica como

consumir o serviço?

Sim, há. Existe uma especificação de um documento chamado WADL (que significa *Web Application Description Language*), que tem por objetivo descrever estes detalhes a respeito do serviço. Este documento ainda não é padronizado, o que quer dizer que nem todos os frameworks REST (de todas as linguagens) o utiliza, e ainda não há um consenso na comunidade a respeito do seu uso. A submissão do formato está disponível em <https://www.w3.org/Submission/wadl/> (HADLEY, 2009).

O Jersey gera este documento automaticamente, e este pode ser visualizado, no nosso caso, ao acessar <http://localhost:8080/cervejaria/services/application.wadl>. Este documento tem o seguinte formato geral:

```
<application xmlns="http://wadl.dev.java.net/2009/02">
    <!-- Esta tag é opcional, e provê documentação a respeito
        da aplicação como um todo -->
    <doc />

    <!-- Esta tag é utilizada para descrever os elementos
        utilizados pelos serviços -->
    <grammars>
        <!-- ... -->
    </grammars>

    <!-- Esta tag é utilizada para começar a descrever os
        serviços -->
    <resources base="http://localhost:8080/cervejaria/services/">

        <!-- Esta tag é utilizada para descrever um serviço
            específico - no caso, o que fica em /cervejas -->
        <resource path="/cervejas">

            <!-- Esta tag descreve uma maneira específica de
                se comunicar com o serviço.
                O atributo name descreve o método a ser
                utilizado -->
            <method id="criarCerveja" name="POST">

                <!-- Descreve os detalhes da requisição -->
                <request>
```

```

<!-- Declara o(s) elemento(s) utilizado(s) na
     requisição. Esta tag pode ser utilizada
     várias vezes -->
<representation element="cerveja"
                 mediaType="text/xml" />
</request>

<!-- Descreve os detalhes da resposta -->
<response>
    <representation />
</response>
</method>
</resource>
</resources>
</application>

```

Portanto, o cliente tem toda a informação necessária a partir do WADL, para que possa acessar corretamente a aplicação.

Vamos analisar, passo a passo, o próprio WADL gerado pelo Jersey para o nosso projeto, começando pela tag de documentação:

```
<doc xmlns:jersey="http://jersey.java.net/"
     jersey:generatedBy="Jersey: 2.4.1 2013-11-08 12:08:47"/>
```

A tag de documentação gerada informa apenas a versão do Jersey e o dia e horário em que foi gerada, sem formato particular.

```

<grammars>
    <include href="application.wadl/xsd0.xsd">
        <doc title="Generated" xml:lang="en"/>
    </include>
</grammars>

```

Esta seção descreve quais XSDs estão sendo usados pela aplicação para descrever as entidades utilizadas. Conforme descrito, se acessarmos <http://localhost:8080/cervejaria/services/application.wadl/xsd0.xsd>, vamos obter o seguinte documento:

```

<?xml version="1.0" standalone="yes"?>
<xss:schema version="1.0"
  xmlns:xss="http://www.w3.org/2001/XMLSchema">

```

```

<xs:element name="cerveja" type="cerveja"/>

<xs:element name="cervejas" type="cervejas"/>

<xs:complexType name="cerveja">
  <xs:sequence>
    <xs:element name="nome"
      type="xs:string" minOccurs="0"/>
    <xs:element name="descricao" type="xs:string"
      minOccurs="0"/>
    <xs:element name="cervejaria" type="xs:string"
      minOccurs="0"/>
    <xs:element name="tipo" type="tipo" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="cervejas">
  <xs:sequence>
    <xs:element name="link" type="jaxbLink" minOccurs="0"
      maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="jaxbLink">
  <xs:sequence/>
  <xs:attribute name="href" type="xs:string"/>
  <xs:anyAttribute namespace="#other"
    processContents="skip"/>
</xs:complexType>

<xs:simpleType name="tipo">
  <xs:restriction base="xs:string">
    <xs:enumeration value="LAGER"/>
    <xs:enumeration value="PILSEN"/>
    <xs:enumeration value="BOCK"/>
    <xs:enumeration value="WEIZEN"/>
  </xs:restriction>
</xs:simpleType>
</xs:schema>

```

A tag seguinte, `resources` , descreve o endereço físico onde estão nossos serviços. Já a posterior, `resource` , descreve o endereço de cada serviço:

```

<resources base="http://localhost:8080/cervejaria/services/">
  <resource path="/cervejas">

```

Note que o valor do atributo `base` é aquele que deve ser usado no método `target` do cliente JAX-RS.

A próxima tag, `method`, provê um mapeamento direto para cada método na classe Java:

```
<method id="criarCerveja" name="POST">
    <request>
        <ns2:representation
            xmlns:ns2="http://wadl.dev.java.net/2009/02"
            xmlns="" element="cerveja" mediaType="text/xml"/>
        <ns2:representation
            xmlns:ns2="http://wadl.dev.java.net/2009/02"
            xmlns="" element="cerveja"
            mediaType="application/xml"/>
        <ns2:representation
            xmlns:ns2="http://wadl.dev.java.net/2009/02"
            xmlns="" element="cerveja"
            mediaType="application/json"/>
    </request>
    <response>
        <representation mediaType="text/xml"/>
        <representation mediaType="application/xml"/>
        <representation mediaType="application/json"/>
    </response>
</method>
```

Perceba que o atributo `id` da tag `method` recebe o nome do método Java. Já o atributo `name` provê mapeamento direto para o método HTTP a ser utilizado.

O restante é mapeado de acordo com o que é declarado na classe. Ou seja, considerando-se que o método `criarCerveja` recebe como parâmetro uma `Cerveja` e retorna `Response`, é possível mapear corretamente o parâmetro, mas não o retorno do método (já que aquilo que é usado pela `Response` é mapeado dinamicamente).

O método seguinte, `listeTodasAsCervejas`, recebe um *query param* como parâmetro:

```
<method id="listeTodasAsCervejas" name="GET">
```

```

<request>
    <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
           name="pagina" style="query" type="xs:int"/>
</request>
<response>
    <ns2:representation
        xmlns:ns2="http://wadl.dev.java.net/2009/02"
        xmlns="" element="cervejas" mediaType="text/xml"/>
    <ns2:representation
        xmlns:ns2="http://wadl.dev.java.net/2009/02"
        xmlns="" element="cervejas"
        mediaType="application/xml"/>
    <ns2:representation
        xmlns:ns2="http://wadl.dev.java.net/2009/02"
        xmlns="" element="cervejas"
        mediaType="application/json"/>
</response>
</method>

```

Note a presença da tag `param`. Ela delimita o nome do parâmetro e seu tipo — de acordo com os tipos delimitados em um XML Schema.

Nem todos os parâmetros são declarados desta forma. Um *path param*, por ser incluído diretamente na URL, recebe a própria seção `resource`:

```

<resource path="{nome}">
    <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
           name="nome" style="template" type="xs:string"/>

```

Assim, pela tag `param`, é feita a associação entre o que é declarado na tag `resource` e a tipagem do parâmetro.

Customizando o WADL

É possível realizar customizações no WADL utilizando a API do Jersey. É claro, também seria possível realizar atualizações manualmente, mas isso incorreria no custo de realizar manutenções constantes nesse documento — o que é indesejado.

Para realizar esta customização, é necessário criar uma classe

que a parametrize. Vamos chamá-la de `WADLConfig` :

```
package br.com.geladaonline.services;

// imports omitidos

public class WADLConfig extends WadlGeneratorConfig {

    @Override
    public List<WadlGeneratorDescription> configure() {

        return new ArrayList<>();
    }
}
```

Repare que esta classe deve estender a classe abstrata `org.glassfish.jersey.server.wadl.config.WadlGeneratorConfig`, que define o método `configure`. Este método será utilizado na customização do nosso WADL.

Para que esta classe seja usada, é necessário informar à *engine* do Jersey. Utilizamos o mecanismo disponível na classe `ApplicationJAXRS` :

```
@Override
public Map<String, Object> getProperties() {
    Map<String, Object> properties = new HashMap<>();
    properties.put("jersey.config.server.provider.packages",
                  "br.com.geladaonline.services");
    properties.put("jersey.config.server.wadl.generatorConfig",
                  WADLConfig.class.getCanonicalName());
    return properties;
}
```

Assim, basta apenas modificar o método `configure` para começar a usar o mecanismo de customização. Este é feito em partes: declara-se uma classe de customização (utilizando o método `generator`) e uma propriedade. Por exemplo, para criarmos uma documentação geral para o serviço, primeiro criamos um arquivo que vai armazenar essa documentação:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```

<applicationDocs
    targetNamespace="http://research.sun.com/wadl/2006/10">
    <doc xml:lang="pt-BR" title="API geladaonline.com.br">
        Esta é a documentação oficial da aplicação
        <a href="http://geladaonline.com.br">
            geladaonline.com.br
        </a>
    </doc>
</applicationDocs>

```

Supondo-se que este arquivo seja chamado de `doc.xml`, nós o referenciamos a partir da propriedade `applicationDocsStream`, da classe `WadlGeneratorApplicationDoc`.

```

public List<WadlGeneratorDescription> configure() {

    return generator(WadlGeneratorApplicationDoc.class)
        .prop("applicationDocsStream", "doc.xml")
        .descriptions();
}

```

Ao acessarmos o WADL novamente, lá está a documentação:

```

<ns2:application xmlns:ns2="http://wadl.dev.java.net/2009/02">
    <ns2:doc xmlns:jersey="http://jersey.java.net/"
        jersey:generatedBy="Jersey: 2.4.1 2013-11-08 12:08:47"/>
    <ns2:doc title="API geladaonline.com.br" xml:lang="en">
        Esta é a documentação oficial da aplicação
        <a href="http://geladaonline.com.br">
            geladaonline.com.br
        </a>
    </ns2:doc>
    <!-- restante do WADL -->
</ns2:application>

```

Também podemos gerar documentação para os métodos da nossa classe. Para isso, geramos um XML com o seguinte formato, por exemplo:

```

<?xml version="1.0" encoding="UTF-8"?>
<resourceDoc>
    <classDocs>
        <classDoc>

```

```

<className>
    br.com.geladaonline.services.CervejaService
</className>
<commentText>
    <![CDATA[Gerencia todas as cervejas do sistema.]]>
</commentText>
<methodDocs>
    <methodDoc>
        <methodName>listeTodasAsCervejas</methodName>
        <commentText>
            <![CDATA[Lista as cervejas catalogadas]]>
        </commentText>
        <responseDoc>
            <representations>
                <representation element="cervejas"
                    status="200" mediaType="text/xml,
                    application/xml,application/json">
                    <doc>Esta é o retorno padrão</doc>
                </representation>
            </representations>
        </responseDoc>
    </methodDoc>
    <methodDoc>
        <methodName>encontreCerveja</methodName>
        <commentText>
            <![CDATA[Localiza uma determinada cerveja
            em particular]]>
        </commentText>
        <responseDoc>
            <representations>
                <representation element="cerveja"
                    status="200" mediaType="text/xml,
                    application/xml,application/json">
                    <doc>Esta é o retorno padrão</doc>
                </representation>
                <representation status="404">
                    <doc>Retornado caso a cerveja em questão
                    não seja encontrada.</doc>
                </representation>
            </representations>
        </responseDoc>
    </methodDoc>
</methodDocs>
<classDoc>
</classDocs>
</resourceDoc>

```

Supondo-se que este XML seja chamado `resource-doc.xml`,

podemos indicar para a engine do Jersey que este documento deve ser utilizado usando a classe `org.glassfish.jersey.server.wadl.internal.generators.resourcedoc.WadlGeneratorResourceDocSupport%`.

```
public List<WadlGeneratorDescription> configure() {  
  
    return generator(WadlGeneratorApplicationDoc.class)  
        .prop("applicationDocsStream", "doc.xml")  
        .generator(WadlGeneratorResourceDocSupport.class)  
        .prop("resourceDocStream", "resourcedoc.xml")  
        .descriptions();  
  
}
```

Isto tem o seguinte efeito, por exemplo:

```
<ns2:method id="encontreCerveja" name="GET">  
  <ns2:doc>  
    Localiza uma determinada cerveja em particular  
  </ns2:doc>  
  <ns2:response status="200">  
    <ns2:representation element="cerveja"  
      mediaType="text/xml,application/xml,application/json">  
      <ns2:doc>  
        This is the representation returned by default  
      </ns2:doc>  
    </ns2:representation>  
  </ns2:response>  
  <ns2:response status="404">  
    <ns2:representation>  
      <ns2:doc>Retornado caso a cerveja em questão não  
      seja encontrada.</ns2:doc>  
    </ns2:representation>  
  </ns2:response>  
</ns2:method>
```

7.5 CONCLUSÃO

Neste capítulo, você aprendeu melhor como funciona REST do ponto de vista do cliente.

O primeiro ponto foi sobre como realizar a interação com

serviços a partir de um browser, utilizando JavaScript e o framework jQuery. Assim, dispensando a necessidade de se construir o *client-side* com mecanismos mais elaborados. O segundo ponto foi sobre como realizar essa interação a partir de um cliente JAX-RS, caso você deseje usar um cliente baseado em padrões.

O terceiro tipo de cliente utilizado foi construído com o RESTEasy, que provê um modelo inovador — baseado em criação de *proxies* dinâmicos a partir de interfaces Java, ou seja, permitindo ao cliente que ele tenha a sensação de estar utilizando algo dentro da própria linguagem (sendo que, na verdade, o serviço não precisa estar implementado em Java).

Finalmente, você conferiu o mecanismo do WADL — um sistema de geração de documentação da API para que seja possível realizar interação com esta sem necessitar de exemplos, ainda que também seja possível fornecê-los.

Vale lembrar de que existem diversas técnicas disponíveis para realizar esta documentação e interação, como é o caso do framework Swagger ou geração automática a partir do WADL, por exemplo. A intenção deste tópico foi apenas ilustrar que existem técnicas disponíveis para documentação de serviços em REST.

Ainda há muito que precisamos ver a respeito de REST, como mecanismos de segurança e tópicos mais avançados, como gerenciamento de concorrência. Vamos em frente?

CAPÍTULO 8

SEGURANÇA REST

"A simplicidade é o último grau de sofisticação." – Leonardo da Vinci

Sua cervejaria está quase concluída. Resta apenas um último detalhe: como controlar a modificação das cervejas, para que possamos impedir qualquer um de criar, atualizar e apagar cervejas conforme a própria vontade?

8.1 SEGURANÇA CONTRA O QUÊ?

Para sabermos como nos proteger, precisamos antes entender por que precisamos de mecanismos de proteção.

Quando realizamos uma requisição HTTP, geralmente esta passa por vários equipamentos de rede: *switches*, *firewalls*, *proxies* etc. Este é o mecanismo padrão de funcionamento da internet, ou seja, sem esses equipamentos, a internet não funcionaria. O problema é que não é possível garantir que todos estes equipamentos são confiáveis, uma vez que é possível adulterar as configurações para que alguém roube essas informações.

Para que você possa visualizar esta situação, execute o comando `tracert` (no Windows) ou `traceroute` (no Linux) para visualizar o fluxo de dados percorridos até a chegada de uma requisição sua aos servidores do Google, por exemplo:

```
alexandre@alexandre-VirtualBox:~$ traceroute www.google.com.br
traceroute to www.google.com.br (173.194.118.31), 30 hops max, 60 byte packets
 1  192.168.1.1 (192.168.1.1)  1.461 ms  1.390 ms  1.359 ms
 2  * * *
 3  187-100-62-21.dsl.telesp.net.br (187.100.62.21)  3.733 ms  3.857 ms  3.826 ms
 4  201-0-5-9.dsl.telesp.net.br (201.0.5.9)  3.793 ms  3.961 ms  3.917 ms
 5  187-100-52-193.dsl.telesp.net.br (187.100.52.193)  4.676 ms 187-100-52-177.dsl.telesp.net.br (187.1
 0.52.177)  67.539 ms 187-100-52-193.dsl.telesp.net.br (187.100.52.193)  4.572 ms
 6  xe-5-1-0-0-grasactm1.red.telefonica-wholesale.net (176.52.253.121)  5.019 ms  4.657 ms  4.585 ms
 7  Google-ae1-0-grasactm1.red.telefonica-wholesale.net (84.16.11.172)  4.561 ms  4.504 ms  4.588 ms
 8  209.85.254.54 (209.85.254.54)  4.307 ms  5.622 ms  5.000 ms
 9  209.85.245.149 (209.85.245.149)  5.492 ms  5.804 ms  6.831 ms
10  gru06s09-tn-f31.ie100.net (173.194.118.31)  5.143 ms  5.381 ms  5.370 ms
alexandre@alexandre-VirtualBox:~$
```

Figura 8.1: Fluxo de dados percorridos

Ou seja, de acordo com a imagem, desde a saída da requisição da minha máquina até a chegada ao servidor do Google, 10 equipamentos diferentes foram rastreados. São poucos. Há casos em que estes equipamentos são, inclusive, injetados artificialmente.

EASTER EGGS FABRICADOS

Caso você seja um fã da série Star Wars, recomendo enviar o comando `tracert -h 97 obiwan.scrye.net` (no Windows), ou `traceroute -m 97 obiwan.scrye.net` (no Linux).

Assim sendo, fica evidente que é possível injetar mecanismos na comunicação com servidores e "roubar" informações. Saiba você, leitor, que este processo sequer é difícil: basta instalar um programa como o Wireshark (disponível em <http://www.wireshark.org/>) e começar a fazer isto na sua própria rede.

Isso dá margem para dois tipos principais de ataques: *man-in-the-middle* e *eavesdropping*.

Eavesdrop significa ouvir escondido, ou seja, é quando o atacante intercepta a requisição, mas não a altera. Pode ser utilizado para interceptar uma requisição simplesmente para saber o conteúdo de uma mensagem (por exemplo, em uma transferência bancária, saber a quantia transferida de uma conta para outra), ou para ter condições de reproduzir a requisição — dando origem ao chamado *replay attack*.

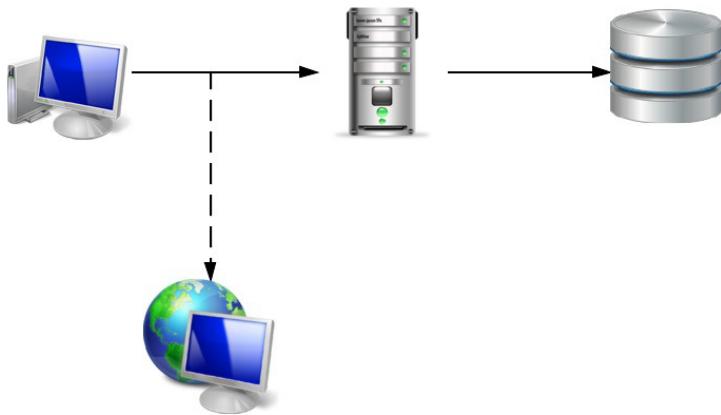


Figura 8.2: Como funciona o eavesdropping

Já o *man-in-the-middle* é ainda pior: o atacante desvia o curso da mensagem, com o propósito de alterá-la e reenviá-la para o servidor, como se tivesse sido enviada pelo remetente legítimo. Isso poderia traduzir-se, no caso de uma transferência bancária, em alterar o destinatário da transferência e o valor.

A proteção contra ambos reside na criptografia: usamos um sistema de proteção forte o suficiente para que, mesmo que o atacante consiga interceptar as mensagens, não consiga alterá-las.

8.2 PROTEÇÃO CONTRA INTERCEPTAÇÃO

COM HTTPS

Em uma comunicação HTTP, o sistema que provê criptografia é o HTTPS (*Hypertext Transfer Protocol over Secure Sockets Layer*), que consiste em aplicar uma camada de segurança com SSL (*Secure Sockets Layer*) sobre o HTTP. Esta camada utiliza certificados digitais para encriptar as comunicações e garantir a autenticidade do servidor e, opcionalmente, do cliente.

Estes certificados são amplamente utilizados na internet. Toda vez que você acessa uma URL cujo endereço começa com `https`, este mecanismo está presente. Por exemplo, ao acessar o GMail (cujo endereço real está presente em <https://mail.google.com>), é possível visualizar as informações do certificado:

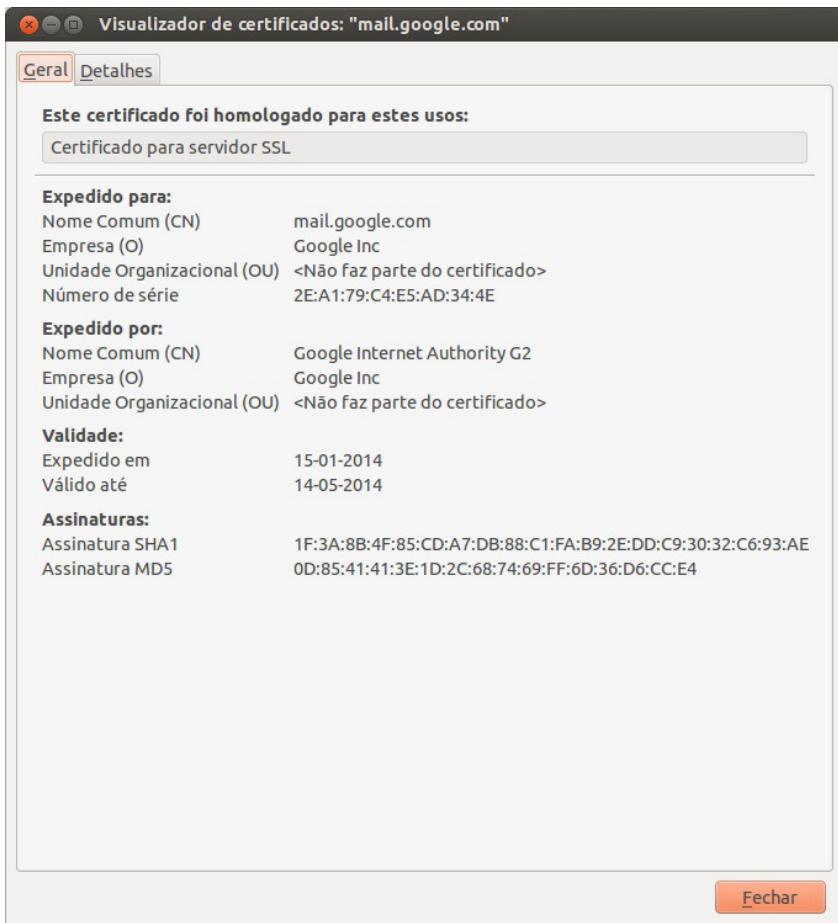


Figura 8.3: Informações do certificado utilizado pelo GMail

Este certificado é usado para que o cliente confirme a autenticidade do servidor (pelas informações contidas na seção **Emitido para**) e criptografe as informações usando esse certificado. Para que o cliente possa ter certeza da confiabilidade desse certificado, ele pode ser emitido por uma autoridade certificadora (*Certification Authority — CA*). Esta autoridade é bem conhecida entre os clientes que confiam nela. Em alguns casos, o certificado também pode ser autoassinado, ou seja, emitido pelo próprio site — o que o torna menos confiável.

Da mesma forma, estes certificados também podem ser levantados pelo cliente para que o servidor confirme a sua identidade — mais ou menos da mesma forma que uma autenticação utilizando usuário e senha. Este modo, no entanto, é mais complicado de ser usado na internet e, em geral, é utilizado apenas em casos muito específicos.

Ao fazer uso de certificados digitais, o servidor provê um mecanismo de criptografia baseado em chaves públicas e privadas. A chave pública é usada para encriptar dados; e a privada, para decriptar. Elas recebem estes nomes porque a ideia geral é que a chave pública seja conhecida por todos os clientes de um determinado serviço, e a privada, apenas pelo provedor do serviço.

A chave pública dos certificados digitais é geralmente utilizada para encriptar um outro tipo de chaves, que são conhecidas como chaves **simétricas**. Estas, por sua vez, são as mesmas tanto para encriptar quanto para decriptar dados. Com o sistema de chave pública e privada, o cliente e o servidor têm a chance de negociar essa chave simétrica — que é usada nas duas pontas para encriptar as mensagens.

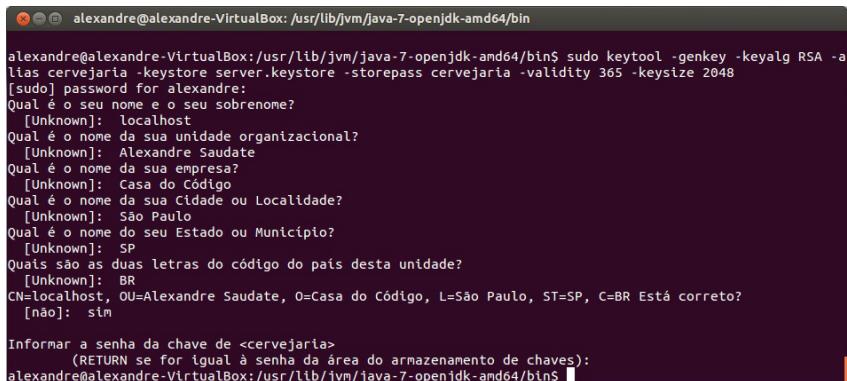
A linguagem Java provê um sistema próprio para armazenamento de certificados, chamado JKS (*Java Key Store*). Um arquivo JKS contém uma coleção de certificados, cada um devidamente identificado por um *alias*. Esses certificados podem ser criados e armazenados utilizando uma ferramenta chamada `keytool`, presente na JDK.

Para utilizar o JKS, você deve navegar até a pasta `bin` da sua JDK, além de possuir uma série de informações, como o domínio da web onde ele será usado, o período durante o qual esse certificado será válido e o algoritmo que será utilizado para encriptar e decriptar seus dados.

Por exemplo, para gerar um certificado válido durante um ano, com algoritmo RSA (com comprimento de chave de 2048 bits) e válido para o endereço `localhost`, o seguinte código pode ser utilizado:

```
keytool -genkey -keyalg RSA -alias knight_usuarios -keystore server.keystore -storepass cervejaria -validity 360  
-keysize 2048
```

Este comando levará a uma série de questionamentos em relação à empresa. Estes dados podem ser populados normalmente, de acordo com as questões, exceto o primeiro (em uma JDK em português, a pergunta é "Qual é o seu nome e sobrenome?"). Neste campo, **deve ser inserido o domínio onde a aplicação será hospedada** — no nosso caso, `localhost`.



```
alexandre@alexandre-VirtualBox:/usr/lib/jvm/java-7-openjdk-amd64/bin$ sudo keytool -genkey -keyalg RSA -alias cervejaria -keystore server.keystore -storepass cervejaria -validity 365 -keysize 2048  
[sudo] password for alexandre:  
Qual é o seu nome e o seu sobrenome?  
[Unknown]: localhost  
Qual é o nome da sua unidade organizacional?  
[Unknown]: Alexandre Saudate  
Qual é o nome da sua empresa?  
[Unknown]: Casa do Código  
Qual é o nome da sua Cidade ou Localidade?  
[Unknown]: São Paulo  
Qual é o nome do seu Estado ou Município?  
[Unknown]: SP  
Quais são as duas letras do código do país desta unidade?  
[Unknown]: BR  
CN=localhost, OU=Alexandre Saudate, O=Casa do Código, L=São Paulo, ST=SP, C=BR Está correto?  
[não]: sim  
Informar a senha da chave de <cervejaria>  
(RETURN se for igual à senha da área de armazenamento de chaves):  
alexandre@alexandre-VirtualBox:/usr/lib/jvm/java-7-openjdk-amd64/bin$
```

Figura 8.4: Questionamentos em relação à criação do certificado

CERTIFICADO AUTOASSINADO

O certificado que criamos é o chamado **certificado autoassinado**. Ele recebe esse nome porque nós mesmos o criamos e o assinamos.

Não há problema algum em usar um certificado autoassinado em tempo de desenvolvimento — mas ele não é recomendado para ambientes de produção, já que não há nenhuma autoridade certificadora que garanta as informações contidas nele.

8.3 IMPLANTANDO SSL NO NOSSO SERVIDOR

Para fazer com que nosso servidor use SSL, é necessário configurá-lo de acordo. Como estamos utilizando um Jetty embarcado no Maven, basta reinserir as informações do plugin do Jetty, com as configurações para utilização de segurança:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>jetty-maven-plugin</artifactId>
      <version>8.1.9.v20130131</version>
      <configuration>
        <scanIntervalSeconds>10</scanIntervalSeconds>
        <stopKey>foo</stopKey>
        <stopPort>9999</stopPort>
        <webApp>
          <contextPath>/cervejaria</contextPath>
        </webApp>
        <loginServices>
          <loginService implementation="org.eclipse.jetty.
            security.HashLoginService">
            <name>Default</name>
          </loginService>
        </loginServices>
      </configuration>
    </plugin>
  </plugins>
</build>
```

```
</loginServices>
<connectors>
    <connector implementation="org.eclipse.jetty
        .server.ssl.SslSocketConnector">
        <port>8443</port>
        <maxIdleTime>60000</maxIdleTime>
        <keystore>
            ${basedir}/src/main/config/server.keystore
        </keystore>
        <password>cervejaria</password>
        <keyPassword>cervejaria</keyPassword>
    </connector>
</connectors>
</configuration>
<executions>
    <execution>
        <id>start-jetty</id>
        <phase>pre-integration-test</phase>
        <goals>
            <goal>start</goal>
        </goals>
        <configuration>
            <scanIntervalSeconds>0</scanIntervalSeconds>
            <daemon>true</daemon>
        </configuration>
    </execution>
    <execution>
        <id>stop-jetty</id>
        <phase>post-integration-test</phase>
        <goals>
            <goal>stop</goal>
        </goals>
    </execution>
</executions>
</plugin>
</plugins>
</build>
```

Note que esta configuração faz referência ao arquivo `server.keystore`, que deve estar em uma nova pasta de código-fonte, chamada `src/main/config`. Este foi gerado pela execução do comando `keytool`, e vai conter o certificado utilizado na criptografia da página.

O QUE, EXATAMENTE, É UM KEY STORE?

O Key Store , em Java, é um arquivo que contém diversos certificados, chaves públicas/privadas etc. Ele recebe esse nome quando usado para encriptar as conexões de saída; porém, também pode ser utilizado para realizar certificação da identidade do cliente — quando passa a ser chamado de Trust Store .

Para realizar o teste, basta executar o servidor normalmente e acessar a URL <https://localhost:8443/cervejaria/services/cervejas>.

8.4 HTTP BASIC

Em aplicações que usam segurança, geralmente é necessário fornecer mecanismos para que o cliente possa se identificar. A partir desta identificação, o servidor checa se as credenciais fornecidas são válidas e quais direitos elas dão ao cliente. Estes procedimentos recebem o nome de autenticação (ou seja, o cliente fornecendo credenciais válidas) e autorização (o servidor checando quais direitos o cliente tem).

O protocolo HTTP oferece dois tipos de autenticação: *basic* e *digest*. A autenticação *basic* é realmente muito simples, consiste em passar o *header* HTTP Authorization obedecendo ao seguinte algoritmo:

```
Basic Base64({#usuário}:{#senha})
```

Considerando usuário alexandre e senha alexandre , o cabeçalho passaria a ter o valor a seguir:

```
Authorization: Basic YWxleGFuZHJl0mFsZXhhbmRyZQ==
```

O ALGORITMO BASE64

O algoritmo **Base64** é amplamente conhecido pela internet. Trata-se de um algoritmo utilizado principalmente para converter dados binários em código alfanumérico.

Para mais informações a respeito, leia <http://tools.ietf.org/html/rfc989>, seção 4.3.

Para que possamos usar autenticação *basic* no Jetty, precisamos utilizar um arquivo contendo os usuários e suas senhas, assim como seus papéis na aplicação. Por exemplo, para definir um usuário admin com senha 123 , e role ADMIN , crie um arquivo `realm.properties` com o seguinte conteúdo:

```
admin:123,ADMIN
```

Salve o arquivo na pasta `src/main/config` do projeto. Referenciamos este arquivo a partir da propriedade config do `loginService` :

```
<loginServices>
    <loginService implementation="org.eclipse.jetty.security.
                                         HashLoginService">
        <name>Default</name>
        <config>
            ${basedir}/src/main/config/realm.properties
        </config>
    </loginService>
</loginServices>
```

Agora, precisamos definir no `web.xml` as URLs que serão protegidas, bem como os métodos HTTP que podem ser utilizados nelas e as *roles* autorizadas.

Isto é feito por meio da inclusão de uma série de declarações no

arquivo `web.xml`. A primeira delas é a inclusão da tag `security-constraint`, que declara qual URL e quais métodos HTTP serão protegidos, e qual *role* de usuário tem direito de acessar os recursos desta URL.

Declaramos, então, que os métodos `GET`, `PUT`, `POST` e `DELETE`, a partir da URL `/services`, só poderão ser acessados a partir de um usuário que tenha a *role* `ADMIN`:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Services</web-resource-name>
        <url-pattern>/services/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
        <http-method>PUT</http-method>
        <http-method>DELETE</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>ADMIN</role-name>
    </auth-constraint>
</security-constraint>
```

Também precisamos declarar o mecanismo de autenticação e o *realm*, isto é, o espaço em que as credenciais são válidas. Para isso, usamos a tag `login-config`:

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

Finalmente, precisamos realizar uma declaração à parte sobre quais *roles* podem ser utilizadas, o que pode ser feito a partir do uso da tag `security-role`:

```
<security-role>
    <role-name>ADMIN</role-name>
</security-role>
```

O conteúdo completo do arquivo fica assim:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
        <servlet-name>
            br.com.geladaonline.services.ApplicationJAXRS
        </servlet-name>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>
            br.com.geladaonline.services.ApplicationJAXRS
        </servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Services</web-resource-name>
            <url-pattern>/services/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
            <http-method>PUT</http-method>
            <http-method>DELETE</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>ADMIN</role-name>
        </auth-constraint>
    </security-constraint>

    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>Default</realm-name>
    </login-config>

    <security-role>
        <role-name>ADMIN</role-name>
    </security-role>
</web-app>

```

Agora, basta realizar o teste. Caso você utilize o `Poster` para realizar esta requisição, deverá ver algo como o seguinte:

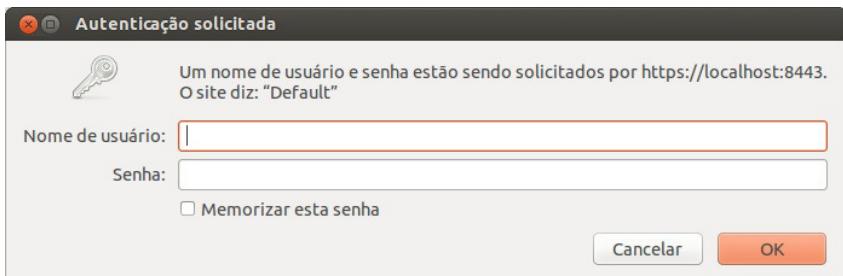


Figura 8.5: Solicitação de autenticação no servidor

8.5 MODIFICANDO O CLIENTE

O próximo passo para esse mecanismo ficar completo é realizar a criação do cliente. Para isso, primeiro criamos um cliente básico para nosso serviço, apontando para o novo endereço, ou seja, <https://localhost:8443/cervejaria/services>:

```
// imports omitidos
public class Client {

    public static void main(String[] args) {

        javax.ws.rs.client.Client client =
            ClientBuilder.newBuilder().build();

        client.register(JettisonFeature.class);

        Cervejas cervejas = client
            .target("https://localhost:8443/cervejaria/services")
            .path("/cervejas")
            .request(MediaType.APPLICATION_JSON)
            .get(Cervejas.class);

        System.out.println(cervejas);

    }
}
```

Se executarmos este código, vamos obter uma exceção com o seguinte *stack trace*:

```

Exception in thread "main" java.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target
        at org.glassfish.jersey.client.HttpURLConnection.apply(HttpURLConnection.java:184)
        at org.glassfish.jersey.client.ClientRuntime.invoke(ClientRuntime.java:227)
        at org.glassfish.jersey.client.JerseyInvocation$2.call(JerseyInvocation.java:671)
        at org.glassfish.jersey.internal.Errors.process(Errors.java:315)
        at org.glassfish.jersey.internal.Errors.process(Errors.java:297)
        at org.glassfish.jersey.internal.Errors.process(Errors.java:285)
        at org.glassfish.jersey.client.ClientHandlerWrapper.invoke(ClientHandlerWrapper.java:110)
        at org.glassfish.jersey.process.internal.RequestScope.runInScope(RequestScope.java:472)
        at org.glassfish.jersey.client.JerseyInvocation.invoke(JerseyInvocation.java:167)
        at org.glassfish.jersey.client.JerseyInvocation$Builder.method(JerseyInvocation.java:396)
        at org.glassfish.jersey.client.JerseyInvocation$Builder.get(JerseyInvocation.java:296)
        at Client.main(Client.java:42)
Caused by: javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target
        at sun.security.ssl.Alert.getSSLException(Alert.java:192)
        at sun.security.ssl.SSLContextImpl.fatal(SSLContextImpl.java:1886)
        at sun.security.ssl.SSLContextImpl.assertValidCertPath(SSLContextImpl.java:1876)
        at sun.security.ssl.Handshaker.fatalSE(Handshaker.java:276)
        at sun.security.ssl.Handshaker.checkCE(Handshaker.java:1340)
        at sun.security.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:1341)
        at sun.security.ssl.ClientHandshaker.processMessage(ClientHandshaker.java:153)
        at sun.security.ssl.Handshaker.processLoop(Handshaker.java:868)
        at sun.security.ssl.Handshaker.process(Handshaker.java:1084)
        at sun.security.ssl.SSLSocketImpl.readRecord(SSLSocketImpl.java:1016)
        at sun.security.ssl.SSLSocketImpl.performInitialHandshake(SSLSocketImpl.java:1312)
        at sun.security.ssl.SSLSocketImpl.startHandshake(SSLSocketImpl.java:1339)
        at sun.security.ssl.SSLSocketImpl.startHandshake(SSLSocketImpl.java:1323)
        at sun.net.www.protocol.https.HttpsClient$HttpsEngine.connect(HttpsClient.java:465)
        at sun.net.www.protocol.https.AbstractDelegateHttpsURLConnection.connect(AbstractDelegateHttpsURLConnection.java:185)
        at sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.java:1299)
        at java.net.HttpURLConnection.getResponseCode(HttpURLConnection.java:468)
        at sun.net.www.protocol.https.HttpsURLConnection.getResponseCode(HttpsURLConnectionImpl.java:338)
        at org.glassfish.jersey.client.HttpURLConnection.apply(HttpURLConnection.java:276)
        at org.glassfish.jersey.client.ClientHttpConnector.apply(HttpURLConnection.java:102)
        ... 18 more
Caused by: sun.security.validator.ValidatorException: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target
        at sun.security.validator.PKIXValidator.doBuild(PKIXValidator.java:385)
        at sun.security.validator.PKIXValidator.engineValidate(PKIXValidator.java:292)
        at sun.security.validator.Validator.validate(Validator.java:388)
        at sun.security.ssl.X509TrustManagerImpl.validate(X509TrustManagerImpl.java:326)
        at sun.security.ssl.X509TrustManagerImpl.checkTrusted(X509TrustManagerImpl.java:231)
        at sun.security.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerImpl.java:126)
        at sun.security.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:1323)
        ... 44 more
Caused by: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target
        at sun.security.provider.certpath.SunCertPathBuilder.engineBuild(SunCertPathBuilder.java:196)
        at java.security.cert.CertPathBuilder.build(CertPathBuilder.java:268)
        at sun.security.validator.PKIXValidator.doBuild(PKIXValidator.java:389)
        ... 30 more

```

Figura 8.6: Exceção

Esta exceção ocorre por conta da falta de um *trust store* registrado, ou seja, uma verificação a respeito da autenticidade do servidor. Para registrar o *trust store*, precisamos criar um novo contexto SSL — algo que pode ser realizado usando-se a classe `org.glassfish.jersey.SslConfigurator`:

```

String baseDir =
    "/home/alexandre/codigoFonte/rest/rest/cap-08-ssl/";

SslConfigurator config = SslConfigurator
    .newInstance()
    .trustStoreFile(baseDir + "/src/main/java/br/com/geladaonline
                           /client/server.keystore")
    .trustStorePassword("cervejaria");

```

A partir desse código, conseguimos criar o contexto SSL (representado pela classe `javax.net.ssl.SSLContext`) e utilizá-lo a partir do `ClientBuilder`:

```

SSLContext context = config.createSSLContext();
javax.ws.rs.client.Client client = ClientBuilder.newBuilder()
    .sslContext(context)

```

```
.build();
```

Agora, a execução deste código deve originar o seguinte *stack trace*:

```
Exception in thread "main" javax.ws.rs.NotAuthorizedException:  
HTTP 401 Unauthorized
```

Neste momento, basta apenas acrescentar a autenticação. A partir da versão 2.5 do Jersey, é possível fazer isso usando a classe:

```
org.glassfish.jersey.client.authentication  
.HttpAuthenticationFeature:  
  
HttpAuthenticationFeature auth =  
    HttpAuthenticationFeature.basic("admin", "123");  
  
client.register(auth);
```

Finalmente, o código deve apresentar a cerveja obtida do servidor.

8.6 HTTP DIGEST

Outro mecanismo de segurança nativo do protocolo HTTP é a autenticação **Digest**. Ela possui uma vantagem sobre a autenticação **Basic**, já que o algoritmo usado para proteção é tão robusto que não necessita de SSL para proteger o usuário e senha (embora ainda precise de SSL para proteger o corpo da requisição). Em compensação, são sempre necessárias duas requisições para que o cliente possa utilizá-la, o que faz este algoritmo ser muitas vezes preferido em favor da autenticação **Basic** com SSL.

Este mecanismo funciona da seguinte maneira: ao efetuar a requisição sem dados de autenticação, o cliente recebe de volta um cabeçalho conhecido como **desafio**. O desafio tem o formato:

```
401 Unauthorized  
WWW-Authenticate: Digest realm="Default",  
qop="auth",
```

```
nonce="0cc194d2c0f4b6b765d448a578773543",
opaque="92ea7e0dd41225f43d534bc544c2223"
```

Note a presença dos campos *realm*, *qop*, *nonce* e *opaque*. Estes representam, respectivamente, o contexto de autenticação (mencionado anteriormente), o nível de proteção oferecido, uma *string* aleatória que será usada no cálculo da autenticação e uma *string*, que deverá ser retornada para o servidor (como prova de que este cabeçalho não foi alterado). O nível de proteção pode ser *auth* — proteção apenas da autenticação —, ou *auth-int* — proteção de integridade da mensagem como um todo.

De posse destes dados, o cliente deve calcular a resposta levando em conta estes dados, além do usuário, senha, número de mensagens já enviadas, método HTTP a ser utilizado, a URL e uma *string* aleatória gerada pelo próprio cliente. Por exemplo, levando-se em conta a resposta anterior para uma requisição GET na URL <https://localhost:8443/cervejaria/services/cervejas>, teríamos o seguinte algoritmo:

```
#método=GET
#path=/cervejaria/services/cervejas

#realm=Default
#nonce=0cc194d2c0f4b6b765d448a578773543
#qop=auth

#contador=00000001
#nonceDoCliente=4a7a0b7765d657a756e2053d2822
#usuário=admin
#senha=123

a1 = HEXADECIMAL(MD5({#usuário}:{#realm}:{#senha}))

// Valor de a1: 30d2058a6fd44ca5b7463cabc6ad4472

a2 = HEXADECIMAL(MD5({#método}:{#path}))

// Valor de a2: 31f544fa002737163b80d7404942c164

a3 = HEXADECIMAL(MD5({a1}:{#nonce}:{#contador}:
{#nonceDoCliente}:{#qop}:{a2}))
```

```
// Valor de a3: 48e051c245f06833a707ac350da94c86
```

O ALGORITMO MD5

O algoritmo **MD5**, assim como Base64, também é amplamente conhecido. No entanto, não é um algoritmo de criptografia, e sim, de *hash*. Por ter essa natureza, uma vez calculado, não pode ser revertido.

Para mais informações, consulte <http://www.ietf.org/rfc/rfc1321.txt>.

Com a variável `a3`, o cliente pode "devolver" a requisição com a devida autorização:

```
GET /services/usuarios HTTP/1.1
Host: localhost
Authorization: Digest username="admin",
realm="Default",
nonce="0cc194d2c0f4b6b765d448a578773543",
uri="/cervejaria/services/cervejas",
qop=auth,
nc=00000001,
cnonce="4a7a0b7765d657a756e2053d2822",
response="48e051c245f06833a707ac350da94c86",
opaque="92ea7eedd41225f43d534bc544c2223"
```

Com estes dados, o servidor calcula o *hash* da mesma forma. Caso a resposta obtida seja igual, o cliente está autorizado; caso contrário, não. Note que o servidor precisa ter conhecimento do usuário e senha (além dos dados passados pelo cliente) para ter condições de calcular esse *hash*. Sem que ele tenha conhecimento antecipadamente, de qual é o usuário e senha, não é possível calcular esse *hash*. Além disso, um possível interceptador da requisição não consegue reenviá-la, já que elementos aleatórios são introduzidos na mensagem e descartados quando usados.

8.7 OAUTH

Certamente, um dos mecanismos mais atraentes a serem utilizados atualmente quando falamos de autenticação/autorização em serviços REST é **OAuth**. Trata-se de uma técnica usada para que o usuário possa reaproveitar seu login e senha de outra aplicação sem fornecer estes dados para a aplicação na qual ele está logando.

O benefício mais óbvio deste sistema é proteger o login e senha de uma aplicação preexistente sem que a aplicação na qual estamos realizando login não conheça, efetivamente, estes dados. Outro benefício é oferecer um acesso **restrito** a estes recursos, ou seja, o usuário consegue visualizar uma lista dos dados aos quais a aplicação está solicitando acesso.

Uma explicação mais didática para isso está presente no blog de um dos criadores da especificação. Ele diz que alguns carros de luxo americanos possuem uma chave especial chamada de *chave para valets*. Baseia-se na ideia de que, se o dono de um desses carros vai a um restaurante, por exemplo, ele vai deixar o carro com um valet, que vai estacionar o carro para ele. No entanto, simplesmente deixar o carro com um desses valets oferece o risco de que o valet faça algo mais com o carro — por exemplo, dar uma volta pela cidade inteira enquanto o proprietário está no restaurante.

Para não correr esse risco, o proprietário entrega a chave de valets, que autoriza o usuário a rodar um limite predefinido de quilômetros e, chegado o limite, o carro não permite que o motorista dirija mais.

A ideia com o OAuth é essencialmente a mesma. Hoje, essa especificação tem duas versões: 1.0 (também chamada de 1.0a) e 2.0. Cada uma tem um fluxo distinto de autenticação / autorização.

No OAuth 1.0, a aplicação cliente – ou seja, a que vai requerer

que o usuário faça login utilizando outra aplicação – possui um registro na aplicação servidora, que fornece à primeira uma chave e uma senha, chamadas neste contexto de **Consumer Key** e **Consumer Secret**. Esses dados podem ser encarados como um login e senha, e devem permanecer em segredo da aplicação.

Quando o usuário solicita acesso à aplicação cliente por meio da aplicação servidora, a cliente vai utilizar essas chaves para solicitar credenciais temporárias e, de posse destas, vai redirecionar o usuário para um link onde este pode realizar seu login. A página para onde o usuário será redirecionado é da aplicação servidora, que o alerta que a aplicação cliente está requisitando o acesso e oferece a listagem de recursos aos quais o acesso será oferecido.

Uma vez que o usuário realizar essa autorização, a aplicação servidora invocará a aplicação cliente de volta, com as credenciais temporárias e informações chamadas de **Access Token** e **Access Token Secret**. Estas podem ser encaradas como o login e senha do usuário, mas com acesso restrito (ou seja, o equivalente à chave do valet). De posse desta chave, a aplicação cliente pode então realizar as requisições em nome do usuário.

Para simplificar o entendimento, assuma você como o usuário, o Twitter como aplicação servidora e nossa cervejaria como aplicação cliente:

1. Realizamos o cadastro no Twitter da nossa aplicação. Será fornecido, portanto, uma *Consumer Key* e uma *Consumer Secret*.
2. Quando você solicita acesso aos seus dados do Twitter pela aplicação da cervejaria, ela vai enviar para o Twitter uma solicitação de token — processo que tem a função de fornecer para a cervejaria uma chave temporária (que vai servir somente como "ponte" entre você e a cervejaria perante o Twitter, mas não vai fornecer acesso a recurso algum).

3. Com essa chave, a cervejaria redireciona você para o Twitter, onde será solicitada autenticação e concessão de acesso aos seus recursos (por exemplo, mensagens do Twitter).
4. Quando você conceder esse acesso, o Twitter invocará a cervejaria de volta, passando como parâmetro uma *string* verificadora.
5. Com essa *string*, a cervejaria invocará novamente o Twitter, passando como parâmetro a *string* verificadora e outros dados, como a *Consumer Key* e a *Consumer Secret* novamente.
6. A resposta desta solicitação será o *Access Token* e o *Access Token Secret*, que darão acesso a estes recursos.

O fluxo completo é o seguinte:

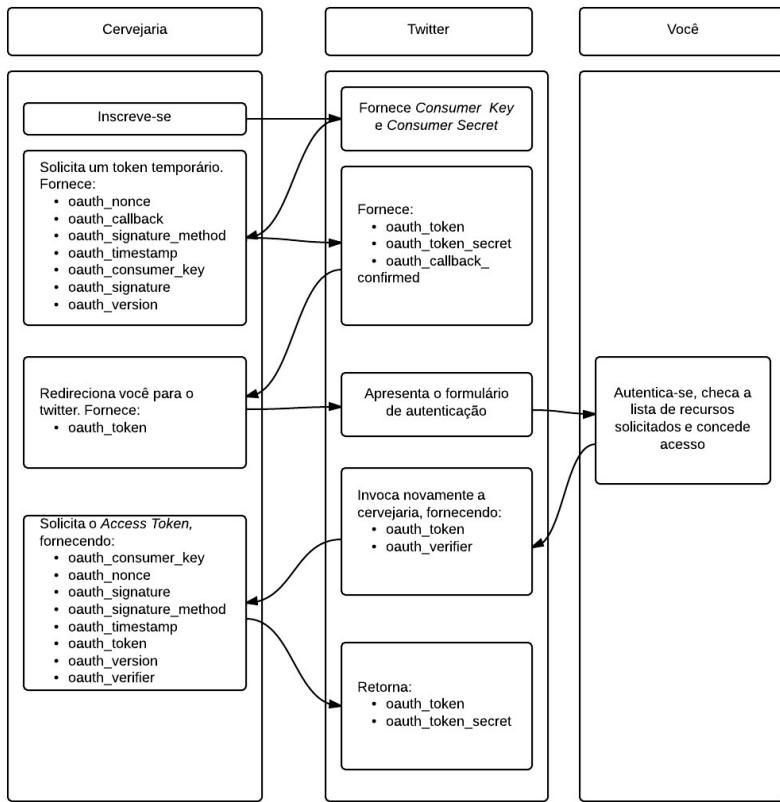


Figura 8.7: Fluxo completo

Os parâmetros envolvidos são:

- **oauth_nonce** : uma *string* aleatória, que será usada na assinatura da mensagem;
- **oauth_callback** : a URL da aplicação cliente;
- **oauth_timestamp** : um *timestamp* da requisição;
- **oauth_consumer_key** : a *Consumer Key* da aplicação cliente;
- **oauth_version** : versão do mecanismo OAuth (neste caso, 1.0);
- **oauth_signature** : um *hash* produzido a partir dos

- dados da mensagem — também chamado de assinatura da mensagem;
- `oauth_signature_method` : o algoritmo utilizado na assinatura da mensagem;
 - `oauth_token` : ora representa um token temporário, ora representa um token definitivo — neste último caso, é também chamado de *Access Token*.
 - `oauth_token_secret` : a senha do token usado;
 - `oauth_verifier` : é o verificador retornado para a aplicação cliente.

Quanto à versão 2 de OAuth, trata-se de uma versão reduzida dos passos envolvidos em OAuth1. É apenas necessário redirecionar o usuário para a aplicação servidora e, quando o acesso é autorizado, a aplicação servidora efetua o *callback* para a aplicação cliente com o *Access Token*. A confiabilidade do protocolo reside totalmente em protocolos de segurança sobre HTTP — razão pela qual todas as requisições realizadas com OAuth 2 precisam utilizar URLs com HTTPS.

Apesar de ser mais simples de ser usado, OAuth 2 propõe menos padronizações sobre o formato geral dos dados envolvidos. Dessa forma, são requeridas especializações do processo no lado do cliente e no lado do servidor, por isso fluxos OAuth 2 são mais difíceis de serem implementados.

8.8 IMPLEMENTANDO ACESSO AO TWITTER ATRAVÉS DO JERSEY

No momento em que escrevo este livro, o Jersey fornece implementações para que se possa criar uma aplicação cliente (para OAuth 1 e 2) ou uma aplicação servidora (somente para OAuth 1).

Vou apresentar nesta seção como realizar o acesso às mensagens

do Twitter via nossa aplicação de cervejaria. Seguindo o fluxo já apresentado, o primeiro passo é realizar o cadastro da aplicação — algo que pode ser feito no site <http://dev.twitter.com>. Uma vez logado, posicione o mouse sobre sua foto e selecione My Applications :



Figura 8.8: Minhas aplicações

A princípio, você deverá ver uma tela mostrando que você não tem nenhuma aplicação criada:

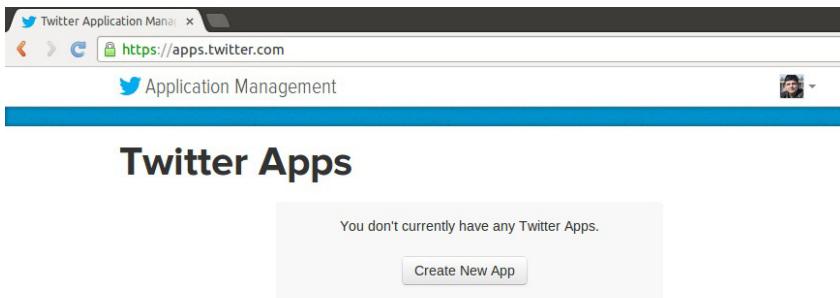


Figura 8.9: Nenhuma aplicação criada

Quando clicar no botão `Create New App`, você será levado ao formulário para criação de uma nova aplicação:

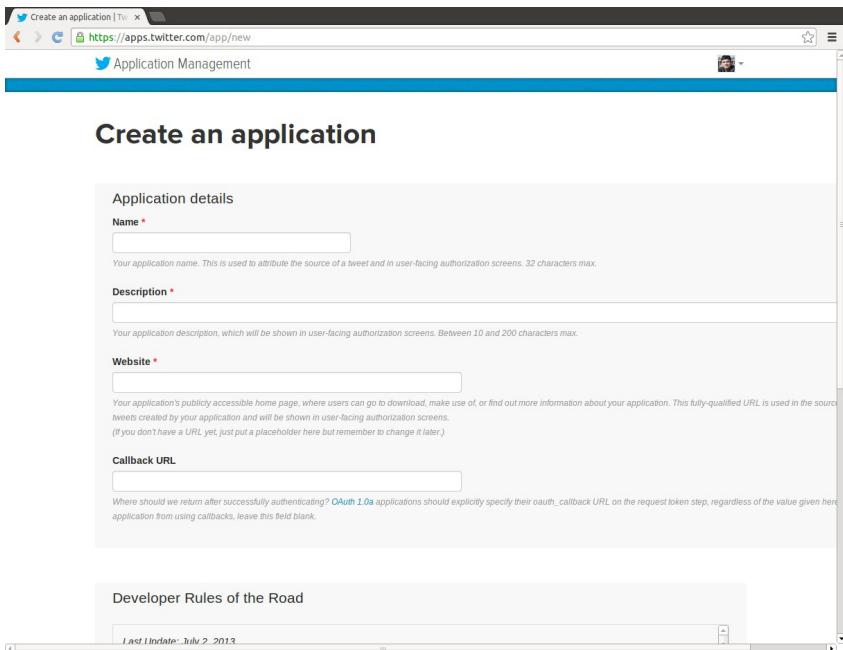


Figura 8.10: Criando uma nova aplicação

Vou criar, então, uma nova aplicação, chamada "Gelada Online". Assim, eu preencho o formulário com Gelada Online no campo **name**, Aplicação de exemplo do livro REST: Construa APIs inteligentes de maneira simples no campo **Description**, e <http://geladaonline.com.br> no campo **Website**.

No campo **Callback URL**, insira o valor <http://127.0.0.1:8080/cervejaria>. Aceite os termos do Twitter e clique em **Create your Twitter Application**. Uma vez feito isto, você deverá ver uma tela como a seguinte:



Gelada Online

[Test OAuth](#)

[Details](#)

[Settings](#)

[API Keys](#)

[Permissions](#)



Aplicação de exemplo do livro REST: Construa API's inteligentes de maneira simples

<http://geladaonline.com.br>

Organization

Information about the organization or company associated with your application. This information is optional.

Organization None

Organization website None

Figura 8.11: App recém-criada

Ao clicar na aba **API Keys**, você obterá os detalhes a respeito do acesso que será concedido para sua aplicação:

Gelada Online

[Test OAuth](#)

[Details](#)

[Settings](#)

[API Keys](#)

[Permissions](#)

Application settings

Keep the "API secret" a secret. This key should never be human-readable in your application.

API key PC0rcbGgL6YhGJ9t8hNXrQ

API secret wWWWHQdQv5dxPjORTLCZFWOpjPMOg2GoHbkU3Ujsxk

Access level Read-only ([modify app permissions](#))

Owner aleaudate

Owner ID 75190230

Application actions

[Regenerate API keys](#)

[Change App Permissions](#)

Figura 8.12: Detalhes do acesso concedido

O que você vê nesta aba são a *Consumer Key* e *Consumer Secret*,

que foram geradas para sua aplicação (chamadas pelo Twitter de **API key** e **API secret**). Observe o alerta gerado nesta própria página: a **API secret** deve permanecer secreta!

Uma vez de posse destes dados, vamos ajustar nosso ambiente para desenvolver nossa aplicação.

O primeiro passo é adicionar as dependências necessárias para o Maven, ou seja, o suporte para clientes OAuth 1 e as dependências necessárias do Jackson (para podermos recuperar *tweets* em formato JSON). O trecho de controle de dependências ficará assim:

```
<properties>
    <jersey.version>2.5.1</jersey.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.glassfish.jersey.core</groupId>
        <artifactId>jersey-server</artifactId>
        <version>${jersey.version}</version>
    </dependency>
    <dependency>
        <groupId>org.glassfish.jersey.containers</groupId>
        <artifactId>jersey-container-servlet</artifactId>
        <version>${jersey.version}</version>
    </dependency>
    <dependency>
        <groupId>org.glassfish.jersey.core</groupId>
        <artifactId>jersey-client</artifactId>
        <version>${jersey.version}</version>
    </dependency>
    <dependency>
        <groupId>org.glassfish.jersey.media</groupId>
        <artifactId>jersey-media-json-jackson</artifactId>
        <version>${jersey.version}</version>
    </dependency>
    <dependency>
        <groupId>org.glassfish.jersey.media</groupId>
        <artifactId>jersey-media-json-jettison</artifactId>
        <version>${jersey.version}</version>
    </dependency>
    <dependency>
        <groupId>org.glassfish.jersey.security</groupId>
        <artifactId>oauth1-client</artifactId>
```

```
<version>${jersey.version}</version>
</dependency>
</dependencies>
```

Também vale notar que, para realizarmos esse procedimento, não precisaremos de SSL (as comunicações entre o usuário e a aplicação da cervejaria não necessitarão de proteção). Assim sendo, podemos desabilitar a segurança, ou usar apenas a porta não segura.

Para desenvolver o código, precisamos de uma abordagem que satisfaça as seguintes restrições:

- O acesso ao código da própria cervejaria não precisa ser restrito por OAuth;
- Precisamos de um mecanismo que armazene o *Access Token* e o *Access Token Secret*, já que não queremos ficar solicitando sempre estes dados do cliente;
- Queremos um código que seja o mínimo intrusivo possível, para que a manutenção seja fácil caso tenhamos muitos serviços que necessitam de autorização OAuth.

Para desenvolver este mecanismo, portanto, criamos as seguintes regras:

- Separamos os recursos que precisam de proteção com algo que possa distingui-los dos outros — por exemplo, acrescentando mais um segmento de URL, como `/twitter/`.
- Para não haver problemas em relação a *clusters*, deixaremos o próprio usuário encarregado de armazenar o *Access Token* e o *Access Token Secret*, por meio de *cookies* — mapas chave-valor que são armazenados em arquivos no lado do cliente e reenviados para o servidor a cada acesso.
- Deixaremos o controle de fluxo em uma classe própria

para controlar o fluxo e interceptaremos as requisições com um filtro — que vai realizar o controle entre deixar o próprio serviço executar a lógica, ou redirecionar o usuário para o Twitter. Este controle vai depender da presença ou não dos *cookies*. Este filtro também deverá controlar o *callback* por parte do Twitter.

A API do Jersey para clientes OAuth é baseada em torno da classe:

```
org.glassfish.jersey.client.oauth1.OAuth1AuthorizationFlow
```

Ela define quais URLs serão usadas em cada passo, bem como a criação de um cliente habilitado para fazer requisições autorizadas por OAuth. Para utilizar o Twitter, por exemplo, usamos o seguinte código:

```
ConsumerCredentials credentials =
    new ConsumerCredentials(CONSUMER_KEY, CONSUMER_SECRET);

OAuth1AuthorizationFlow flow = OAuth1ClientSupport
    .builder(credentials)
    .authorizationFlow(
        "https://api.twitter.com/oauth/request_token",
        "https://api.twitter.com/oauth/access_token",
        "https://api.twitter.com/oauth/authorize")
    .build();

String authorizationUri = flow.start();
```

Para criar o fluxo de autorização, utilizamos a classe:

```
org.glassfish.jersey.client.oauth1.OAuth1ClientSupport
```

As URLs passadas como parâmetro para o método `authorizationFlow` são a de requisição do *token* temporário, de obtenção do *Access Token* e de autorização do usuário, respectivamente. Também passamos a *Consumer Key* e *Consumer Secret* como parâmetro por meio do método `builder` (encapsulados na classe

```
org.glassfish.jersey.client.oauth1.ConsumerCredentials )
```

Finalmente, quando invocamos o método `start`, uma URI é gerada. Devemos redirecionar o usuário para essa URI para que ele possa realizar a autenticação. Note que ainda falta informar qual a URL de *callback*.

O *callback* informado pode ser a própria URL que foi invocada pelo usuário, isto é, a URL do serviço da cervejaria que estamos invocando. Para isso, precisamos recuperar a URL invocada e também ajustá-la no fluxo OAuth.

Para recuperar a URL, usamos a interface `javax.servlet.http.HttpServletRequest`, método `getRequestURL`:

```
private static final String CONSUMER_KEY =
    "PC0rcbGgL6YhGJ9t8hNxRQ";
private static final String CONSUMER_SECRET
    = "wWWHQdQv5dxPj0RTLZFWOpjPM0g2GoHbkU3UjSxk";

public static String init(HttpServletRequest req) {

    StringBuffer callback = req.getRequestURL();
    String callbackHost = callback.toString()

    ConsumerCredentials credentials = new ConsumerCredentials(
        CONSUMER_KEY, CONSUMER_SECRET);

    OAuth1AuthorizationFlow flow = OAuth1ClientSupport
        .builder(credentials)
        .authorizationFlow(
            "https://api.twitter.com/oauth/request_token",
            "https://api.twitter.com/oauth/access_token",
            "https://api.twitter.com/oauth/authorize")
        .callbackUri(callbackHost)
        .build();

    String authorizationUri = flow.start();
}
```

ONDE ARMAZENAR A CONSUMER KEY E CONSUMER SECRET?

Recomendo fortemente que você armazene as chaves da aplicação em local externo ao código. Isso porque estes podem ser encarados como um login e uma senha, então, eles podem ser alterados entre ambientes (desenvolvimento / homologação / produção) e também diretamente pelo site do Twitter.

Temos de cuidar de certas restrições com este código. A primeira é que a busca por mensagens pode conter parâmetros (*query strings*). A segunda é que o próprio Twitter não permite que o *callback* seja efetuado para uma URL.

Agora, vamos criar o recurso que terá o acesso controlado por OAuth. Como dito antes, ele deverá ter uma URL padronizada para ser interceptada por um filtro. Assim, vou criar um serviço que liste as mensagens do Twitter:

```
package br.com.geladaonline.services;

import java.util.List;

import javax.servlet.http.HttpServletRequest;
import javax.ws.rs.*;
import javax.ws.rs.client.Client;
import javax.ws.rs.core.*;

import org.glassfish.jersey.client.oauth1.AccessToken;

import br.com.geladaonline.model.twitter.*;

@Path("/twitter/messages")
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
public class TwitterMessages {

    @GET
    public Statuses list(HttpServletRequest req) {
```

```

        Client client = ClientBuilder.newClient()
            .register(JacksonFeature.class);

        Response response = client.target(
            "https://api.twitter.com/1.1/statuses"
                "/home_timeline.json").request()
            .get();

        Status[] statusList = response.readEntity(Status[].class);

        Statuses status = new Statuses();

        status.setStatusCollection(Arrays.asList(statusList));

        return status;

    }
}

```

Note que a interface `javax.servlet.http.HttpServletRequest` vem diretamente do Servlet do JAX-RS que recebeu a requisição e a repassou para nosso serviço. Para obter esse acesso, no entanto, precisamos de uma anotação especial: `javax.ws.rs.core.Context`. Através dela, obtemos acesso a vários recursos que estarão presentes no contexto da invocação aos serviços.

Utilizamos a anotação diretamente no parâmetro, assim:

```
public Statuses list(@Context HttpServletRequest req)
```

Note que também estamos utilizando um modelo próprio para o Twitter, com as classes `Status`, `Statuses` e `User`. Como o modelo do próprio Twitter é um tanto extenso, vamos fazer com que nossa *engine* descarte propriedades que não queremos visualizar por meio da anotação `org.codehaus.jackson.annotate.JsonIgnoreProperties`. As classes estão descritas a seguir:

```
package br.com.geladaonline.model.twitter;
```

```

import javax.xml.bind.annotation.*;
import org.codehaus.jackson.annotate.JsonIgnoreProperties;

@XmlRootElement
@JsonIgnoreProperties(ignoreUnknown = true)
public class Status {

    @XmlElement(name = "created_at")
    private String createdAt;
    @XmlElement(name = "text")
    private String text;
    @XmlElement(name = "user")
    private User user;

    public String getCreatedAt() {
        return createdAt;
    }

    public String getText() {
        return text;
    }

    public User getUser() {
        return user;
    }
}

```

A modificação na classe `Statuses` fica assim:

```

package br.com.geladaonline.model.twitter;

import java.util.*;
import javax.xml.bind.annotation.*;

@XmlRootElement
public class Statuses {

    private List<Status> statusCollection = new ArrayList<>();

    @XmlElement(name="status")
    public List<Status> getStatusCollection() {
        return statusCollection;
    }

    public void
    setStatusCollection(List<Status> statusCollection) {
        this.statusCollection = statusCollection;
    }
}

```

```
    }
}
```

E a modificação na classe User :

```
package br.com.geladaonline.model.twitter;

import javax.xml.bind.annotation.*;
import org.codehaus.jackson.annotate.JsonIgnoreProperties;

@XmlRootElement
@JsonIgnoreProperties(ignoreUnknown = true)
public class User {

    @XmlElement(name = "name")
    private String name;

    public String getName() {
        return name;
    }
}
```

Vamos agora à implementação do filtro de acesso. Para isso, vamos criar um novo filtro e fazê-lo interceptar somente as requisições para o serviço de mensagens:

```
package br.com.geladaonline.services;

import java.io.IOException;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.*;

@WebFilter("/services/twitter/*")
public class TwitterLoginFilter implements Filter {

    public void doFilter(ServletRequest request,
                         ServletResponse response, FilterChain chain)
        throws IOException, ServletException {

        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse resp = (HttpServletResponse) response;
    }

    //outros métodos da interface Filter
}
```

Note que, a cada requisição para os serviços que comecem com a URL `/services/twitter`, o método `doFilter` será invocado. Ou seja, o conteúdo desse método deverá checar se os *cookies* que contêm o *Access Token* e o *Access Token Secret* (que serão agrupados pela classe

`org.glassfish.jersey.client.oauth1.AccessToken`) foram fornecidos e, caso contrário, deverá checar se a requisição é, na verdade, um *callback* do Twitter.

Caso nenhum dos dois casos seja verdadeiro, deverá inicializar o fluxo de autenticação do OAuth, redirecionando o usuário para a URL de autorização do Twitter em vez de deixar a requisição chegar até o serviço. O código do método, de maneira geral, fica assim:

```
import org.glassfish.jersey.client.oauth1.AccessToken;

public class TwitterLoginFilter implements Filter {

    public static final String OAUTH_VERIFIER = "oauth_verifier";
    public static final String OAUTH_TOKEN_FIELD = "oauth_token";

    public static final String ACCESS_TOKEN_KEY = "AccessToken";

    public void doFilter(ServletRequest request,
                         ServletResponse response, FilterChain chain)
        throws IOException, ServletException {

        AccessToken accessToken =
            recuperarAccessTokenDosCookies(req);

        if (accessToken == null) {

            //Estes parâmetros serão fornecidos caso a requisição
            //seja o callback
            String verifier = req.getParameter(OAUTH_VERIFIER);
            String token = req.getParameter(OAUTH_TOKEN_FIELD);

            if (verifier != null && token != null) {
                //Ainda resta criar o método de verificação
                accessToken =
                    TwitterOAuthFlowService.verify(token,
                                                    verifier, req);
            }
        }
    }
}
```

```

        ajustaCookiesNaResposta(resp, accessToken);
    } else {
        //Redirecionamos o usuário para URL fornecida pelo
        //Twitter
        String twitterAuthUri =
            TwitterOAuthFlowService.init(req);
        resp.sendRedirect(twitterAuthUri);
        return;
    }
}

//Ajusta o Access Token na requisição para que possamos
//utilizá-lo posteriormente
req.setAttribute(ACCESS_TOKEN_KEY, accessToken);
chain.doFilter(req, resp);
}
}

```

A partir deste método, precisamos implementar vários métodos. Comecemos pelo ajuste do *Access Token* em formato de *cookies* para o cliente:

```

public static final String TOKEN_COOKIE = "TwitterAccessToken";
public static final String TOKEN_COOKIE_SECRET =
    "TwitterAccessTokenSecret";

private void ajustaCookiesNaResposta(HttpServletRequest resp,
    AccessToken accessToken) {

    Cookie accessTokenCookie = new Cookie(TOKEN_COOKIE,
                                           accessToken.getToken());
    accessTokenCookie.setPath("/");

    Cookie accessTokenSecretCookie =
        new Cookie(TOKEN_COOKIE_SECRET,
                   accessToken.getAccessTokenSecret());
    accessTokenSecretCookie.setPath("/");

    resp.addCookie(accessTokenCookie);
    resp.addCookie(accessTokenSecretCookie);
}

```

Agora, veremos como recuperar esses *cookies* a partir da requisição do cliente:

```
protected AccessToken recuperaAccessTokenDosCookies(
```

```

        HttpServletRequest req) {

    String accessToken = null;
    String accessTokenSecret = null;

    //Itera pelos cookies fornecidos pelo usuário
    Cookie[] cookies = req.getCookies();
    for (Cookie cookie : cookies) {
        if (cookie.getName().equals(TOKEN_COOKIE)) {
            accessToken = cookie.getValue();
        } else if (cookie.getName().equals(TOKEN_COOKIE_SECRET)) {
            accessTokenSecret = cookie.getValue();
        }
    }

    if (accessToken != null && accessTokenSecret != null) {
        return new AccessToken(accessToken, accessTokenSecret);
    }
    return null;
}

```

Finalmente, veremos como realizar a verificação. Porém, existe aqui um problema: a requisição inicial do cliente para o serviço é uma; o *callback* do Twitter e o repasse para o serviço, outra. Só que a instância criada pela classe `OAuth1AuthorizationFlow` **precisa** ser a mesma. Como resolver esse problema?

Podemos ajustar esta classe na sessão do usuário e, então, fazer com que o Twitter nos informe qual é a sessão partindo do seu ID. Isto pode ser resolvido a partir do parâmetro `jsessionid`, que vamos ajustar como URL de *callback* no método `init` da classe `TwitterOAuthFlowService`. Também aproveitamos para ajustar eventuais parâmetros que o usuário tenha passado para o serviço:

```

StringBuffer callback = req.getRequestURL();
callback.append(";jsessionid=");
    .append(req.getSession()
        .getId());
if (req.getQueryString() != null) {
    callback.append("?").append(req.getQueryString());
}

String callbackHost = callback.toString()

```

```
.replace("localhost", "127.0.0.1");
```

Agora que temos uma maneira de ligar as sessões, ajustamos a instância de `OAuth1AuthorizationFlow` na sessão pelo parâmetro gerado `oauth_token` (que será retornado pelo *callback* do Twitter). O método completo fica assim:

```
private static final String OAUTH_TOKEN_FIELD = "oauth_token";

public static String init(HttpServletRequest req) {

    ConsumerCredentials credentials = new ConsumerCredentials(
        CONSUMER_KEY, CONSUMER_SECRET);

    StringBuffer callback = req.getRequestURL();
    callback.append(";jsessionid=")
        .append(req.getSession()
            .getId());
    if (req.getQueryString() != null) {
        callback.append("?").append(req.getQueryString());
    }

    String callbackHost =
        callback.toString().replace("localhost", "127.0.0.1");

    OAuth1AuthorizationFlow flow = OAuth1ClientSupport
        .builder(credentials)
        .authorizationFlow(
            "https://api.twitter.com/oauth/request_token",
            "https://api.twitter.com/oauth/access_token",
            "https://api.twitter.com/oauth/authorize")
        .callbackUri(callbackHost)
        .build();

    String authorizationUri = flow.start();

    String token = authorizationUri.substring(authorizationUri
        .indexOf(OAUTH_TOKEN_FIELD) +
        OAUTH_TOKEN_FIELD.length() + 1);

    req.getSession().setAttribute(token, flow);

    return authorizationUri;
}
```

Finalmente, implementamos o método de verificação da

autenticidade do *callback*. Esta verificação é feita a partir do método `finish` da classe:

```
org.glassfish.jersey.client.oauth1.OAuth1AuthorizationFlow.
```

Lembre-se de que, para recuperarmos sua instância, utilizamos o próprio token recebido como resposta do Twitter:

```
public static AccessToken verify(String token, String verifier,
    HttpServletRequest req) {

    OAuth1AuthorizationFlow flow = (OAuth1AuthorizationFlow) req
        .getSession().getAttribute(token);

    AccessToken accessToken = flow.finish(verifier);

    req.getSession().removeAttribute(token);

    return accessToken;

}
```

O fluxo das invocações, no primeiro acesso, é o seguinte:

1. O método `doFilter`, em `TwitterLoginFilter`, é invocado;
2. O método `recuperaAccessTokenDosCookies` é invocado, mas retorna `null`;
3. Os parâmetros `oauth_verifier` e `oauth_token` não estarão presentes;
4. O método `init` da classe `TwitterOAuthFlowService` será invocado;
5. Isso fará com que seja criada uma instância da classe `OAuth1AuthorizationFlow`, que terá o método `start` invocado (retornando, assim, a URL para onde o usuário deverá ser redirecionado). Além disso, esta instância será armazenada na sessão HTTP do usuário;
6. Uma vez redirecionado para esta URL, o usuário deverá ver um formulário de autenticação e também de autorização da

- aplicação a acessar seus dados;
7. Quando o usuário faz esta autorização, será realizado um *callback* com destino à URL <http://127.0.0.1:8080/cervejaria/services/twitter/messages> (que foi a URL que foi invocada pelo usuário em primeiro lugar);
 8. Este *callback* fará com que o método `doFilter` seja executado novamente;
 9. O método `recuperaAccessTokenDosCookies` será invocado novamente, mas continuará retornando `null`;
 10. Os parâmetros `oauth_verifier` e `oauth_token` estarão presentes. Assim sendo, o método `verify` na classe `TwitterOAuthFlowService` será invocado;
 11. A instância de `OAuth1AuthorizationFlow` responsável por gerenciar o fluxo será recuperad1. da sessão;
 12. O valor do parâmetro `oauth_verifier` será passado como parâmetro para o método `finish`;
 13. A invocação deste método vai retornar uma instância de `org.glassfish.jersey.client.oauth1.AccessToken` ;
 14. Esta instância terá seus dados armazenados como *cookies*;
 15. Esta instância também será ajustada na requisição para que os métodos do serviço possam recuperá-lo.

A partir do Access Token , o serviço poderá invocar o Twitter. O passo final, no entanto, é preparar o cliente para que seja possível realizar estas requisições. Isto é feito a partir da classe `OAuth1ClientSupport` , que gera uma instância de `javax.ws.rs.core.Feature` . Esta instância é ajustada no cliente que, a partir de então, fica preparado para realizar requisições utilizando OAuth:

```
public static Client getClient(AccessToken accessToken) {  
    ConsumerCredentials credentials = new ConsumerCredentials(  
        CONSUMER_KEY, CONSUMER_SECRET);  
  
    Feature feature =
```

```

OAuth1ClientSupport.builder(credentials).feature()
.accessToken(accessToken).build();

client client = ClientBuilder.newClient().register(feature)
.register(JacksonFeature.class);

return client;

}

```

Agora, alteramos o método `list` na classe `TwitterMessages`, para que obtenha o cliente do Twitter a partir deste método:

```

@GET
public Statuses list(@Context HttpServletRequest req) {

    AccessToken token = (AccessToken) req
        .getAttribute(TwitterLoginFilter.ACCESS_TOKEN_KEY);

    Client client = TwitterOAuthFlowService.getClient(token);

    Response response = client
        .target("https://api.twitter.com/1.1/statuses
               /home_timeline.json")
        .request().get();

    Status[] statusList = response.readEntity(Status[].class);

    Statuses status = new Statuses();
    status.setStatusCollection(Arrays.asList(statusList));

    return statuses;
}

```

Está quase concluído! Já podemos realizar o teste de funcionamento, basta acessar a URL <http://localhost:8080/cervejaria/services/twitter/messages> — o que deverá provocar um redirecionamento para o serviço de autenticação do Twitter:

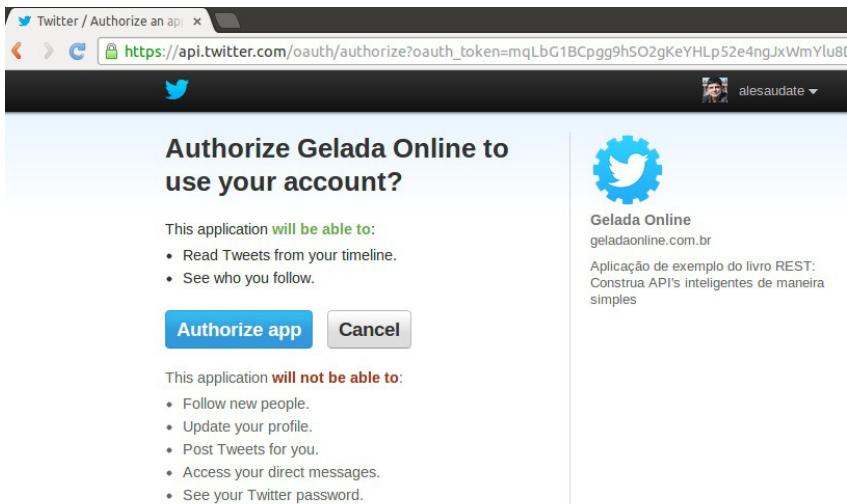


Figura 8.13: Autenticação

Uma vez autenticado, ele fará o redirecionamento:

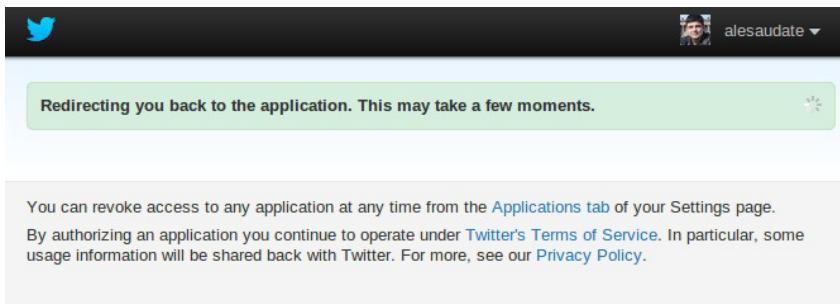


Figura 8.14: Redirecionamento

Finalmente, teremos os dados atualizados:

```

<statuses>
  <status>
    <created_at>Wed Mar 05 00:44:14 +0000 2014</created_at>
    <text>
      O [REDACTED] e eu vimos (finalmente) Dallas Buyers Club e os oscars p/ os atores foi bastante merecido.
      Emocionante e revoltante!
    </text>
    <user>
      <name>[REDACTED]</name>
    </user>
  </status>
  <status>
    <created_at>Wed Mar 05 00:43:01 +0000 2014</created_at>
    <text>
      Kuala Lumpur, Malaysia, 1964 http://t.co/j3xPkNlnS
    </text>
    <user>
      <name>History In Pictures</name>
    </user>
  </status>
  <status>
    <created_at>Wed Mar 05 00:42:15 +0000 2014</created_at>
    <text>
      [REDACTED] essa galera definitivamente não mora e não recebe como alguém que vive no RJ/SP. Não mesmo! Senão,
      deve ser muito caro viver
    </text>
    <user>
      <name>[REDACTED]</name>
    </user>
  </status>
  <status>
    <created_at>Wed Mar 05 00:40:33 +0000 2014</created_at>
    <text>
      GridGain Goes Open Source Under Apache v2.0 - http://t.co/Mb5W2U9lF5 - @DZone Big Link by nivanov
    </text>
    <user>
      <name>DZone</name>
    </user>
  </status>
  <status>
    <created_at>Wed Mar 05 00:37:07 +0000 2014</created_at>
  </status>

```

Figura 8.15: Dados atualizados

Um último ponto a ser levado em consideração é que o usuário é livre para revogar os *Access Tokens* fornecidos a qualquer momento (no caso do Twitter, isto é feito na página <https://twitter.com/settings/applications>). No entanto, os *cookies* ainda permanecerão na máquina do usuário caso isso seja realizado. O que fazer, então?

Caso o usuário faça isso, API do Twitter vai fazer com que o código 401 (Unauthorized) seja retornado. Ou seja, tudo o que temos a fazer é interceptar este erro e reinicializar o processo de autorização OAuth, além de limpar os valores dos *cookies*.

Este procedimento será implementado por meio de um novo método na classe `TwitterOAuthFlowService` :

```
public static
```

```

String reissueAuthorization(HttpServletRequest req,
HttpServletRequest resp ) {

    String redirectURL = init(req);

    Cookie accessTokenCookie =
        new Cookie(TwitterLoginFilter.TOKEN_COOKIE,
        TwitterLoginFilter.EMPTY_COOKIE);
    accessTokenCookie.setPath("/");

    Cookie accessTokenSecretCookie = new Cookie(
        TwitterLoginFilter.TOKEN_COOKIE_SECRET,
        TwitterLoginFilter.EMPTY_COOKIE);
    accessTokenSecretCookie.setPath("/");

    resp.addCookie(accessTokenCookie);
    resp.addCookie(accessTokenSecretCookie);

    return redirectURL;
}

```

Agora, modificaremos o método do serviço para que ele reaja a esta situação. Note que teremos de modificar a própria assinatura do método para isso:

```

@GET
public Response list(@Context HttpServletRequest req,
    @Context HttpServletResponse resp) throws URISyntaxException {

    AccessToken token = (AccessToken) req
        .getAttribute(TwitterLoginFilter.ACCESS_TOKEN_KEY);

    Client client = TwitterOAuthFlowService.getClient(token);

    try {
        Response response = client
            .target("https://api.twitter.com/1.1/statuses"
            "/home_timeline.json")
            .request().get();

        Status[] statusList = response.readEntity(Status[].class);
        Statuses status = new Statuses();

        status.setStatusCollection(Arrays.asList(statusList));

        return Response.ok(status).build();
    }
}

```

```

} catch (WebApplicationException ex) {
    if (ex.getResponse().getStatus() ==
        Response.Status.UNAUTHORIZED.getStatusCode()) {

        String twitterAuthUri = TwitterOAuthFlowService
            .reissueAuthorization(req, resp);

        return
            Response.temporaryRedirect(new URI(twitterAuthUri))
            .build();

    } else {
        throw ex;
    }
}
}

```

8.9 CONSTRUINDO UM SERVIDOR OAUTH

Sua aplicação cliente do Twitter funciona. Mas e se você quiser construir uma aplicação que esteja do outro lado, isto é, e se você quiser construir uma aplicação servidora OAuth?

Felizmente, a API do Jersey também nos auxilia com isso. Vamos fazer com que nossa cervejaria aceite autenticação OAuth para acessar nossos serviços.

Comecemos pelo usual: adicionando as dependências do Maven. Adicione, então, as dependências do OAuth (lado servidor) e da API do Apache Commons IO (que será útil mais tarde):

```

<dependency>
    <groupId>org.glassfish.jersey.security</groupId>
    <artifactId>oauth1-server</artifactId>
    <version>${jersey.version}</version>
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.4</version>
</dependency>

```

No JAR oauth1-server estão contidos serviços que

automatizam o fornecimento da chave temporária e da definitiva. Só precisamos habilitar o reconhecimento e implantação destes — algo que é feito a partir da propriedade `ENABLE_TOKEN_RESOURCES`, presente na classe:

```
org.glassfish.jersey.server.oauth1.OAuth1ServerProperties:
```

```
public Map<String, Object> getProperties() {  
    Map<String, Object> properties = new HashMap<>();  
    properties.put("jersey.config.server.provider.packages",  
        "br.com.geladaonline.services");  
    properties.put(OAuth1ServerProperties.ENABLE_TOKEN_RESOURCES,  
        Boolean.TRUE);  
    return properties;  
}
```

O próximo passo é adicionar um provedor de dados ao contexto do JAX-RS. Isto será feito a partir da instanciação de uma classe que implemente a interface

`org.glassfish.jersey.server.oauth1.OAuth1Provider`. O Jersey contém uma implementação padrão, que mantém estes dados em memória. Esta classe é `org.glassfish.jersey.server.oauth1.DefaultOAuth1Provider`; é necessário instanciá-la e, se você assim desejar, preenche-la com dados de aplicações consumidoras:

```
public Set<Object> getSingletons() {  
  
    DefaultOAuth1Provider provider = new DefaultOAuth1Provider();  
    String IDDoConsumidor = "App consumidora";  
    String consumerKey = "123";  
    String consumerSecret = "123";  
  
    provider.registerConsumer(IDDoConsumidor, consumerKey,  
        consumerSecret,  
        new org.glassfish.jersey.internal.util  
            .collection.MultivaluedStringMap());  
  
    // Restante do método  
}
```

Agora, passamos o provedor como parâmetro para o construtor

da classe:

```
org.glassfish.jersey.server.oauth1.OAuth1ServerFeature  
.OAuth1ServerFeature
```

Ele vai efetivamente habilitar o uso de OAuth na nossa aplicação:

```
Set<Object> singletons = new HashSet<>();  
singletons.add(new JettisonFeature());  
singletons.add(new OAuth1ServerFeature(provider));  
  
return singletons;
```

Uma vez incluída a *feature* OAuth no contexto da nossa aplicação, o Jersey vai cuidar de automatizar o gerenciamento dos tokens, mas não vai fornecer o formulário de autorização. A criação deste deve ser feita de forma manual. Vamos, portanto, criar um formulário simples para isso:

```
<html>  
  <body>  
    <form  
      action="https://localhost:8443/cervejaria/services  
              /authorize"  
      method="post">  
  
      <input type="hidden" name="oauth_token"  
             value="$$$OAUTH_TOKEN$$$" />  
      <label>Usuário:</label><br>  
      <input type="text" name="username" /><br>  
  
      <label>Senha:</label><br>  
      <input type="password" name="password" /><br>  
      <input type="submit" value="Autorizar aplicação" />  
    </form>  
  </body>  
</html>
```

Note que esta página é tão somente um *template*. Posteriormente, vamos alterar o valor do *placeholder* \$\$\$OAUTH_TOKEN\$\$\$ com o valor do token OAuth para que possamos efetuar o *callback* de maneira correta.

Agora, vamos criar o serviço que vai servir este formulário. Na verdade, a ideia é de que ele vá além disso: nosso serviço REST vai atender ao método `GET` (quando ele receber a invocação por parte do cliente) e ao método `POST` (quando o cliente confirmar a autorização da aplicação cliente). Vamos começar, no entanto, pela simples leitura do formulário de autorização, que simplificaremos pelo uso da biblioteca Commons IO:

```
package br.com.geladaonline.services;

import static javax.ws.rs.core.MediaType.*;
import javax.ws.rs.*;
import org.apache.commons.io.IOUtils;

@Path("/authorize")
public class AuthorizeOAuth {

    private static String AUTHORIZE_FORM;

    static {
        try {
            AUTHORIZE_FORM = IOUtils.toString(AuthorizeOAuth.class
                .getResourceAsStream("/login.html"));
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Agora, vamos tratar de criar os métodos, a começar pelo que vai responder ao início do processo de autorização. Lembre-se de que nosso serviço vai preparar um formulário para o cliente e, assim, é interessante adaptar o serviço como um todo para responder a formulários; já o resultado do método `GET` vai ser um HTML:

```
@Path("/authorize")
@Consumes(APPLICATION_FORM_URLENCODED)
public class AuthorizeOAuth {

    @GET
    @Produces(TEXT_HTML)
```

```

public String beginAuthorization(
        @QueryParam("oauth_token") String token)
throws URISyntaxException {

    String htmlForm =
        AUTHORIZE_FORM.replace("$$$OAUTH_TOKEN$$$", token);

    return htmlForm;

}

```

Para fazer com que este formulário apareça, é simples: basta acessar https://localhost:8443/cervejaria/services/authorize?oauth_token=teste (sim, esse processo deve ser feito com HTTPS) e conferir a página:



Figura 8.16: Formulário

Agora, o próximo passo é criar o retorno do método. Este método deverá recuperar o token e informar ao provedor OAuth de que a requisição foi alterada. Em uma aplicação real, também deverá checar se o usuário e senha informados estão corretos. Para que possamos receber estes dados corretamente do formulário, vamos usar *form params*:

```

@POST
public Response authorizeApp(@FormParam("oauth_token")
        String token, @FormParam("username") String usuario,
        @FormParam("password") String senha)
throws URISyntaxException {

```

Para termos acesso ao provedor OAuth, vamos recorrer novamente à anotação `@Context`:

```
@Context  
private DefaultOAuth1Provider provider;
```

Para recuperar o token original, utilizamos o método `getRequestToken` no provedor:

```
@POST  
public Response authorizeApp(@FormParam("oauth_token")  
    String token,  
    @FormParam("username") String usuario,  
    @FormParam("password") String senha)  
throws URISyntaxException {  
  
    final Token requestToken = provider.getRequestToken(token);
```

Agora, vamos invocar o método `authorizeToken` para informar ao provedor de que a requisição foi autorizada e de que ele deve gerar um verificador. Este método recebe três parâmetros: o token original, que recuperamos a partir do método `getRequestToken`; uma implementação da interface `java.security.Principal`, que vai representar o cliente perante o serviço; e um mapa contendo as *roles* que este consumidor terá. Assim sendo, a autorização fica assim:

```
String verifier =  
    provider.authorizeToken(requestToken, new Principal(){  
  
        public String getName() {  
            return provider.getConsumer(  
                requestToken.getConsumer().getKey()).getOwner();  
        }  
    }, roles);
```

POR QUE UTILIZAR A INTERFACE JAVA.SECURITY.PRINCIPAL?

Ao usarmos esta anotação, o framework está implicitamente nos permitindo utilizar outras *features* de segurança do Java, como por exemplo a anotação `javax.annotation.security.RolesAllowed`.

Agora, só temos de preparar o redirecionamento por parte do cliente:

```
//O cliente nos informou qual era a URL de callback quando
//solicitou o token temporário
StringBuilder callbackUrl =
    new StringBuilder(requestToken.getCallbackUrl());

//Se a URL de callback continha parâmetros, estes devem ser
//respeitados
if (callbackUrl.toString().contains("?")) {
    callbackUrl.append("&");
}
else {
    callbackUrl.append("?");
}

callbackUrl.append("oauth_verifier=").append(verifier)
    .append("&");
callbackUrl.append("oauth_token=").append(token);

//O browser do cliente vai efetuar o redirecionamento quando
//receber o código 302 Found
return Response.status(Status.FOUND)
    .location(new URI(callbackUrl.toString())).build();
```

Falta um último toque: acessar, pelo código da aplicação, qual o usuário atual do serviço, ou seja, qual a aplicação cliente. Para isso, injetamos a interface `javax.ws.rs.core.SecurityContext` no nosso serviço e, a partir dela, podemos recuperar o usuário (`Principal`):

```
@Context
private SecurityContext context;

@GET
@Path("{nome}")
public Cerveja encontreCerveja(@PathParam("nome")
String nomeDaCerveja){

    Principal principal = context.getUserPrincipal();
    String nomeDoUsuario = null;

    if (principal != null) {
        nomeDoUsuario = principal.getName();
    }
}
```

```
System.out.println("Quem está acessando? " + nomeDoUsuario);
```

Assim, a parte servidora está concluída. Vale lembrar de que, no mundo real, provavelmente você não vai querer confiar em uma implementação de registro em memória. Pode ser interessante modificar a implementação para trabalhar com um servidor de armazenamento rápido de dados, como o Memcached (<http://memcached.org/>). Para fazer isso, basta implementar a interface

```
org.glassfish.jersey.server.oauth1.OAuth1Provider
```

 e substituí-la nos pontos onde você trabalha com a implementação padrão.

Para realizar os testes, criei uma aplicação cliente que é um *proxy* dos serviços da cervejaria. Esta aplicação é bastante semelhante à aplicação cliente do Twitter, então, para fins de brevidade, não vou realizar o detalhamento desta aqui.

Se você quiser detalhes sobre esta, confira o repositório de código-fonte deste livro: <https://github.com/alesaudate/rest>.

8.10 CONCLUSÃO

Neste capítulo, você conferiu uma série de detalhes a respeito de segurança em serviços REST. Você pôde ver em detalhes as necessidades de se utilizar segurança, bem como os mecanismos usados para isso. Você pôde visualizar como criar e usar um certificado autoassinado para utilização em HTTPS, e como realizar autenticação e autorização de serviços com algoritmos Basic e Digest.

Você também pôde conferir como um dos protocolos mais populares da atualidade, o OAuth, é combinado entre duas aplicações para compor uma solução de autorização elegante e

robusta. Você também viu como implementar um cliente e um servidor prontos para utilizar OAuth.

Agora, resta saber como resolver alguns desafios complexos do mundo real. Vamos em frente?

CAPÍTULO 9

TÓPICOS AVANÇADOS DE SERVIÇOS REST

"Seja lá o que você fizer, seja bom nisso." – Abraham Lincoln

Até aqui, vimos algumas das coisas que fazem uma API REST ser **boa**. Neste capítulo, quero mostrar algumas das coisas que fazem uma API ser excelente, apresentando alguns obstáculos que, por mais corriqueiros que sejam, ainda geram grandes problemas para pessoas que querem implementar suas APIs.

Os tópicos estão divididos aqui nesse capítulo por seção e cada um pode ser adotado de forma independente, assim, você também pode ir direto à seção que mais saltar aos seus olhos e depois voltar para ler os outros tópicos.

9.1 BUSCA POR EXEMPLOS

Um dos maiores desafios que enfrentamos quando escrevemos serviços REST é a modelagem. Muitas vezes, os casos que temos em mãos não é possível de ser escrito utilizando integralmente as regras de modelagem REST.

Mesmo assim, ainda há casos em que é possível, mas muitas pessoas não veem como realizar. Um desses casos é a famosa busca por exemplos, ou seja, fornecemos ao serviço um subconjunto de informações que serão usadas na busca dos nossos dados, e o

serviço deve retornar as entidades que satisfazem àquele critério.

Esse é um caso que geralmente confunde implementadores de APIs REST, porque estes não conseguem ver como modelar este caso utilizando o método `GET`, já que, quando usamos esse método, precisamos modelar todos os parâmetros da busca por exemplos como *query strings*.

Quando utilizamos um banco de dados, isto normalmente se traduz em alterar a consulta ao banco (com construções como a *Criteria* do Hibernate). Já que não estou usando banco de dados neste livro, farei aqui uma implementação artificial, baseada nos dados que possuo em memória. Assim sendo, preciso que a classe `Cerveja` faça a identificação do exemplo, algo que pretendo fazer usando expressões regulares:

```
//imports

import java.util.regex.Pattern;
public class Cerveja {
    //atributos, getters e setters, etc.

    public boolean matchExemplo(Cerveja cerveja) {
        boolean match = true;
        match &= matchRegex(cerveja.nome, this.nome);
        match &= matchRegex(cerveja.descricao, this.descricao);
        match &= matchRegex(cerveja.cervejaria, this.cervejaria);
        match &= this.tipo != null ?
            matchRegex(cerveja.tipo.name(), this.tipo.name())
            : true;

        return match;
    }

    private boolean matchRegex(String toCompare, String source) {
        if (source != null) {
            return Pattern.compile(source).matcher(toCompare).find();
        }
        return true;
    }
}
```

Observe que o método `matchRegex` executa a análise de campos baseado em expressões regulares. O método `matchExemplo` fornece os dados para aquele exemplo, e vai jogando o resultado da análise na variável `match`. Caso alguma das comparações retorne `false`, então, o resultado da implementação retornará `false`.

O próximo passo é construir o exemplo. Para isso, vamos criar um facilitador, baseado no *design pattern* `Builder` (JOHNSON; VLISSIDES; GAMMA; HELM, 1994):

```
public class Cerveja {  
  
    //restante da classe  
  
    public static Builder builder() {  
        return new Builder();  
    }  
  
    public static class Builder {  
  
        private Cerveja building;  
  
        public Builder() {  
            building = new Cerveja();  
            building.nome = "";  
            building.descricao = "";  
            building.cervejaria = "";  
            building.tipo = null;  
        }  
  
        public Builder withNome(String nome) {  
            building.nome = nome;  
            return this;  
        }  
  
        public Builder withDescricao(String descricao) {  
            building.descricao = descricao;  
            return this;  
        }  
  
        public Builder withCervejaria(String cervejaria) {  
            building.cervejaria = cervejaria;  
            return this;  
        }  
    }  
}
```

```

        public Builder withTipo(Tipo tipo) {
            building.tipo = tipo;
            return this;
        }

        public Builder withTipo(String tipo) {
            if (tipo == null || tipo.trim().equals("")) {
                return this;
            }
            building.tipo = Tipo.valueOf(tipo);
            return this;
        }

        public Cerveja build() {
            return building;
        }
    }
}

```

Para conseguir extrair os *query params* da requisição de maneira dinâmica, utilizamos uma interface especial, chamada `javax.ws.rs.core.UriInfo`. Esta pode ser injetada diretamente na instância usando a anotação `javax.ws.rs.core.Context`.

Feito isto, invocamos o método `getQueryParameters`, que vai retornar uma instância da interface `javax.ws.rs.core.MultivaluedMap`. Ela é semelhante a um mapa comum, mas em vez de armazenar um mapa chave-valor, cada chave leva a vários valores diferentes. Com este mapa, criaremos um novo método na classe `Estoque`, `listarCervejasPorExemplos`:

```

public List<Cerveja> listarCervejasPorExemplos(
    int numeroPagina,
    int tamanhoPagina,
    MultivaluedMap<String, String> exemplos) {

    List<Cerveja> resultados = new ArrayList<>(tamanhoPagina);

    Cerveja exemplo = Cerveja.builder()
        .withNome(exemplos.getFirst("nome"))
        .withCervejaria(exemplos.getFirst("cervejaria"))
        .withDescricao(exemplos.getFirst("descricao"))

```

```

        .withTipo(exemplos.getFirst("tipo"))
        .build();

    for (Cerveja cerveja : listarCervejas()) {
        if (exemplo.matchExemplo(cerveja)) {
            resultados.add(cerveja);
        }
    }

    return filtrarPaginacao(numeroPagina, tamanhoPagina,
                           resultados);
}

```

Finalmente, precisamos do código que vai receber estes dados do cliente. Para isso, alteramos a implementação do método `listeTodasAsCervejas`, de forma que ele também faça essa busca por exemplos. A implementação fica assim:

```

public class CervejaService {

    @Context
    private UriInfo uriInfo;

    @GET
    public Cervejas listeTodasAsCervejas(
        @QueryParam("pagina") int pagina) {

        MultivaluedMap<String, String> queryMap =
            uriInfo.getQueryParameters();

        List<Cerveja> cervejas = estoque.
            listarCervejasPorExemplos(pagina, TAMANHO_PAGINA,
                                       queryMap);

        return new Cervejas(cervejas);
    }
}

```

Para testar, basta abrir um browser e invocar a URL de listagem, passando como parâmetro parte de qualquer dos dados de uma cerveja:



A screenshot of a web browser window. The address bar shows 'localhost:8080/cervejaria/services/cervejas?nome=Erdinger'. The page content displays an XML document with one item:

```
<?xml version="1.0"?>
<cervejas>
  <link href="cervejas/Erdinger%20Weissbier" title="Erdinger Weissbier" rel="cerveja"/>
</cervejas>
```

Figura 9.1: Resultado de uma busca por exemplos

9.2 TRANSFORMAÇÃO DE FUNÇÕES EM REST — VALIDAÇÃO DE CPF/CNPJ

Um ponto que costuma ser particularmente complexo em REST é a transformação de coisas que, tradicionalmente, são funções. Elas geralmente oferecem desafios, porque não estamos acostumados a modelá-las como recursos, e não porque são complicadas.

Vou tomar um exemplo: uma simples validação de CPF ou CNPJ. Como estamos falando de um serviço REST, precisamos começar com o "tripé" de serviços REST: definir a URL, o método HTTP e o tipo de dados.

Começando pela URL, podemos separá-la em duas partes: **validacao** e **cpf** — assim, deixamos em aberto para incluir outros tipos de validação.

O próximo ponto é o método HTTP. Como queremos recuperar o resultado de uma validação, utilizamos o método `GET`. Como a descrição do resultado não necessariamente nos interessa nesse caso (apenas o código de retorno), podemos também usar o método `HEAD`.

Finalmente, definimos o tipo de dados. Considero seguro trabalhar com texto plano, ou seja, `text/plain`.

O próximo ponto a ser definido é o mecanismo de parametrização. Tenha em mente que temos, aqui, dois tipos de

informação que precisamos fornecer: um é o CPF, propriamente dito, e o outro é o algoritmo a ser usado. Costuma-se utilizar *path params* quando o parâmetro é obrigatório, e *query params* quando o parâmetro é opcional. Sendo assim, passamos o CPF como *path param* e o algoritmo como *query param*, o que vai gerar uma URL com o seguinte formato:

```
/validacao/cpf/12345678909?algoritmo=MODULO11
```

Vamos, então, à implementação. A primeira coisa que vamos criar é um enum de apoio, onde podemos colocar os tipos de algoritmos utilizados:

```
package br.com.geladaonline.services;

public enum AlgoritmoValidacao {

    MODULO_11 {
        public boolean validar(String cpf) {
            return cpf.equals("12345678909");
        }
        public String getNomeAlgoritmo() {
            return "Módulo 11";
        }
    },
    RECEITA {
        public boolean validar(String cpf) {
            return cpf.equals("12345678909");
        }

        public String getNomeAlgoritmo() {
            return "Checagem na Receita Federal";
        }
    },
    TODOS {
        public boolean validar(String cpf) {
            return MODULO_11.validar(cpf) && RECEITA.validar(cpf);
        }

        public String getNomeAlgoritmo() {
            return MODULO_11.getNomeAlgoritmo() + " e "
                + RECEITA.getNomeAlgoritmo();
        }
    };
}

public abstract boolean validar(String cpf);
```

```
    public abstract String getNomeAlgoritmo();  
  
}
```

O próximo passo é o serviço. Sabendo as regras que acabamos de ver, a implementação é simples:

```
//imports omitidos  
  
@Path("/validacao")  
@Produces(MediaType.TEXT_PLAIN)  
@Consumes(MediaType.TEXT_PLAIN)  
public class ValidacaoService {  
  
    @Path("/cpf/{valor}")  
    @GET  
    public String validarCPF(  
        @PathParam("valor") String cpf,  
        @QueryParam("algoritmo") AlgoritmoValidacao algoritmo){  
        boolean resultado = algoritmo.validar(cpf);  
        if (!resultado) {  
            throw new WebApplicationException(Response  
                .status(Status.BAD_REQUEST)  
                .entity("CPF inválido de acordo com o algoritmo "  
                    + algoritmo.getNomeAlgoritmo()).build());  
        }  
        return "CPF válido";  
    }  
  
    @Path("/cpf/{valor}")  
    @HEAD  
    public Response validarCPFSemResultado(  
        @PathParam("valor") String cpf,  
        @QueryParam("algoritmo") AlgoritmoValidacao algoritmo){  
        validarCPF(cpf, algoritmo);  
        return Response.ok().build();  
    }  
}
```

Aqui, resta apenas um passo: se o algoritmo é opcional, por que não estamos tomando o devido cuidado para o caso de o cliente deixar de fornecer este valor?

Nós vamos usar uma anotação que resolve o caso:

`javax.ws.rs.DefaultValue`. Ao utilizar esta anotação, a engine do JAX-RS atribui um valor padrão ao parâmetro, caso o cliente não o forneça. Tomemos como exemplo o caso do método anotado com `HEAD`:

```
@Path("/cpf/{valor}")
@HEAD
public Response validarCPFSemResultado(
    @PathParam("valor") String cpf,
    @QueryParam("algoritmo") @DefaultValue("TODOS")
        AlgoritmoValidacao algoritmo) {
    validarCPF(cpf, algoritmo);
    return Response.ok().build();
}
```

Portanto, caso o cliente não passe nenhum valor como o *query param* `algoritmo`, automaticamente a engine atribui o valor `TODOS` ao parâmetro. O mesmo pode ser realizado com vários tipos de parâmetro que são objetos, como `Integer`, `Long` e `String`.

9.3 TRANSFORMAÇÃO DE FUNÇÕES EM REST — ENVIO DE E-MAILS

Um caso mais complexo de transformações de funções em recursos é um envio de e-mails, devido às restrições envolvidas:

- O corpo de um e-mail pode ser texto plano ou HTML;
- Um e-mail pode conter vários anexos, de tipos diversos.

Vamos então, realizar a definição de acordo com o exposto na seção anterior:

- **URL:** `/email`;
- **Método:** `POST` (estamos "criando" um novo e-mail no servidor);
- **Tipos de dados:** `text/plain` ou `text/html`, de acordo com o corpo da requisição.

Assim sendo, vamos utilizar o corpo da requisição para passar a mensagem. Mas e em relação a outras informações, como a mensagem, destinatário etc.?

Encaramos essas outras informações como *metadados* e, assim, passamos estas como **cabeçalhos** HTTP. Assim sendo, de certa forma, estamos emulando o próprio protocolo SMTP (para envio de e-mails). Para definirmos estes cabeçalhos, usamos a anotação `javax.ws.rs.HeaderParam`:

```
//imports omitidos

@Path("/email")
public class EmailService {

    @POST
    @Consumes({ TEXT_PLAIN, TEXT_HTML })
    public void enviarEmailSimples(@HeaderParam("To") String para,
        @HeaderParam("Cc") String comCopia,
        @HeaderParam("Bcc") String comCopiaOculta,
        @HeaderParam("Subject") String assunto,
        String mensagem) {

        //implementação
    }
}
```

Além disso, também precisamos recuperar o tipo de conteúdo. Poderíamos fazer isto por meio da interface `HttpServletRequest`. Porém, existe uma ainda mais apropriada para recuperarmos cabeçalhos: a interface `javax.ws.rs.core.HttpHeaders`, que injetamos via anotação `@Context`:

```
@POST
@Consumes({ TEXT_PLAIN, TEXT_HTML })
public void enviarEmailSimples(@HeaderParam("To") String para,
    @HeaderParam("Cc") String comCopia,
    @HeaderParam("Bcc") String comCopiaOculta,
    @HeaderParam("Subject") String assunto,
    String mensagem,
    @Context HttpHeaders httpHeaders) {

    //implementação
}
```

```
}
```

Vamos modelar uma entidade para o nosso e-mail. Para isso, vamos desenhar a classe com uma **API fluente**, ou seja, uma que requer pouco tratamento por parte do usuário (incluindo tratamento de valores nulos). Classes que seguem modelagem de APIs fluentes geralmente têm métodos que, em vez de serem *setters* comuns, começam com a palavra `with`. Eles retornam uma instância da classe, em vez de `void`, para que o usuário não tenha de repetir a declaração da variável utilizada:

```
public class Email {  
  
    private String mensagem = "";  
  
    private String formatoMensagem = FORMATO_PADRAO;  
  
    private String assunto = "";  
  
    private List<String> destinatarios = new ArrayList<>();  
  
    private List<String> comCopia = new ArrayList<>();  
  
    private List<String> comCopiaOculta = new ArrayList<>();  
  
    public static final String SEPARADOR_ENDERE COS = ",";  
  
    public static final String FORMATO_PADRAO = "text/plain";  
  
    public Email withMensagem(String mensagem, String formato) {  
        if (mensagem != null && formato != null) {  
            this.mensagem = mensagem;  
            this.formatoMensagem = formato;  
        }  
        return this;  
    }  
  
    public Email withAssunto(String assunto) {  
        if (assunto != null) {  
            this.assunto = assunto;  
        }  
        return this;  
    }  
}
```

```

public Email withDestinatario(String destinatario) {
    if (destinatario != null) {
        withDestinatarios(destinatario
                            .split(SEPARADOR_ENDEREOS));
    }
    return this;
}

public Email withDestinatarios(String... destinatarios) {
    if (destinatarios != null) {
        this.destinatarios.addAll(Arrays.asList(destinatarios));
    }
    return this;
}

public Email withComCopia (String comCopia) {
    if (comCopia != null) {
        withComCopias(comCopia.split(SEPARADOR_ENDEREOS));
    }
    return this;
}

public Email withComCopias(String... comCopias) {
    if (comCopias != null) {
        this.comCopia.addAll(Arrays.asList(comCopias));
    }
    return this;
}

public Email withComCopiaOcultta (String comCopiaOcultta) {
    if (comCopiaOcultta != null) {
        withComCopiasOcultas(
            comCopiaOcultta.split(SEPARADOR_ENDEREOS));
    }
    return this;
}

public Email withComCopiasOcultas(String... copiasOcultas) {
    if (copiasOcultas != null) {
        this.comCopiaOcultta
            .addAll(Arrays.asList(copiasOcultas));
    }
    return this;
}

@Override
public String toString() {
    return "Email [mensagem=" + mensagem + ", "

```

```

        formatoMensagem=" + formatoMensagem + ", assunto="
        + assunto + ", destinatarios=" + destinatarios + ",
        comCopia=" + comCopia + ", comCopiaOculta="
        + comCopiaOculta + ", anexos=" + anexos + "]";
    }

    public void enviar() {
        System.out.println("Enviando email...: " + toString());
    }

}

```

Agora, tudo o que temos a fazer é instanciar esta classe e passar os dados do e-mail como parâmetro:

```

@POST
@Consumes({ TEXT_PLAIN, TEXT_HTML })
public void enviarEmailSimples(@HeaderParam("To") String para,
    @HeaderParam("Cc") String comCopia,
    @HeaderParam("Bcc") String comCopiaOculta,
    @HeaderParam("Subject") String assunto,
    String mensagem,
    @Context HttpHeaders httpHeaders) {

    Email email = new Email()
        .withDestinatario(para)
        .withComCopia(comCopia)
        .withComCopiaOculta(comCopiaOculta)
        .withAssunto(assunto)
        .withMensagem(mensagem, httpHeaders.getMediaType()
            .toString());

    email.enviar();
}

}

```

Mas e em relação aos anexos?

Existe uma técnica no protocolo HTTP que nos permite trafegar uma série de conteúdos de diversos tipos em uma mesma mensagem HTTP. Esta técnica é chamada de *multipart HTTP*, e consiste de enviar a requisição com um delimitador. Esse delimitador é responsável por "quebrar" a requisição em várias partes, que vão conter seus próprios cabeçalhos (e tipos), neste

formato:

```
POST /email HTTP/1.1
Host: localhost:8080
Content-Type: multipart/mixed; boundary=Delimitador
Content-Length: 582

--Delimitador
Content-Type: text/plain

Mensagem do email
--Delimitador
Content-Type: application/pdf
Conteúdo binário de um PDF
```

Observe a declaração do cabeçalho: `Content-Type`. Além de ter o valor `multipart/mixed`, ela conta com o atributo `boundary`, que declara qual será o texto, dentro do HTTP, que será responsável por separar cada conteúdo. A cada ocorrência de uma sequência de dois traços (-) mais o delimitador, é interpretado que se trata de um novo trecho de HTTP, cada um com seu próprio tipo de informação.

Assim sendo, para enviar um e-mail, podemos colocar a mensagem propriamente dita em um trecho do HTTP, e os anexos em trechos separados. Dessa forma, passamos vários anexos como parâmetro.

Para fazer isso, precisamos incluir o suporte do Jersey a `multipart`, com a seguinte instrução Maven:

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-multipart</artifactId>
    <version>${jersey.version}</version>
</dependency>
```

O próximo passo é realizar a inclusão de `multipart` na aplicação. Para isso, voltamos à declaração da classe `ApplicationJAXRS` e acrescentamos uma instância de `org.glassfish.jersey.media.multipart.MultiPartFeature`:

```
public Set<Object> getSingletons() {  
    Set<Object> singletons = new HashSet<>();  
    singletons.add(new JettisonFeature());  
    singletons.add(new MultiPartFeature());  
  
    return singletons;  
}
```

Finalmente, modelamos o método que faz a utilização da requisição *multipart*. Para fazer isso, substituímos o parâmetro `String conteudo` pela classe `org.glassfish.jersey.media.multipart.MultiPart`:

```
@POST  
@Consumes("multipart/mixed")  
public void enviarEmailAnexos(  
    @HeaderParam("To") String para,  
    @HeaderParam("Cc") String comCopia,  
    @HeaderParam("Bcc") String comCopiaOculta,  
    @HeaderParam("Subject") String assunto,  
    MultiPart multiPart) {  
}
```

Esta classe possui o método `getBodyParts`, que retorna uma lista de instâncias da classe `org.glassfish.jersey.media.multipart.BodyPart`. Cada uma dessas instâncias representa um pedaço da requisição HTTP. Assim sendo, podemos usá-la para iterar sobre os pedaços e ir agrupando os anexos. Mas antes, precisamos modelar uma classe de anexos e incluir esta classe na classe `Email`:

```
public class Anexo {  
  
    private byte[] conteudo;  
  
    private String tipoDeConteudo;  
  
    public Anexo() {}  
  
    public Anexo(byte[] conteudo, String tipoDeConteudo) {  
        this.conteudo = conteudo;  
        this.tipoDeConteudo = tipoDeConteudo;  
    }  
}
```

```

        public String toString() {
            return "Anexo [conteudo=" + Arrays.toString(conteudo)
                + ", tipoDeConteudo=" + tipoDeConteudo + "]";
        }
    }

    public class Email {
        // atributos e métodos definidos anteriormente

        private List<Anexo> anexos = new ArrayList<>();

        public Email withAnexo (Anexo anexo) {
            if (anexo != null) {
                this.anexos.add(anexo);
            }
            return this;
        }
    }
}

```

Agora, implementamos o método para inclusão dos anexos:

```

Email email = new Email()
    .withDestinatario(para)
    .withAssunto(assunto)
    .withComCopia(comCopia)
    .withComCopiaOculta(comCopiaOculta);
for (BodyPart bodyPart : multiPart.getBodyParts()) {
    String mediaType = bodyPart.getMediaType().toString();
    if (mediaType.startsWith("text/plain") ||
        mediaType.startsWith("text/html")) {
        email = email.withMensagem(bodyPart
            .getEntityAs(String.class),
            mediaType);
    }
    else {
        Anexo anexo =
            new Anexo(bodyPart.getEntityAs(byte[].class),
            bodyPart.getMediaType().toString());
        email = email.withAnexo(anexo);
    }
}

email.enviar();

```

Para testar o envio de e-mails simples, basta utilizar o método `header` para preencher os cabeçalhos customizados:

```
ClientBuilder  
    .newClient()  
    .target("http://localhost:8080/cervejaria/services/email")  
    .request()  
    .header("To", "alesaudate@gmail.com")  
    .header("Cc", "fake@fake.com")  
    .header("Bcc", "fake2@fake.com")  
    .header("Subject", "Olá, mundo!")  
    .post(Entity  
        .text("Teste de envio de mensagens de email utilizando  
        JAX-RS"));
```

Isto deve provocar a seguinte saída de texto no console do servidor:

```
Enviando email...: Email [mensagem=Teste de envio de mensagens  
de email utilizando JAX-RS, formatoMensagem=text/plain,  
assunto=Olá, mundo!, destinatarios=[alesaudate@gmail.com],  
comCopia=[fake@fake.com], comCopia0 culta=[fake2@fake.com],  
anexos=[]]
```

Em relação ao teste com anexos, o teste é bastante similar. O que muda é que temos de instanciar a classe `Multipart`. Uma vez instanciada, usamos o método `bodyPart` para acrescentar trechos da requisição HTTP, desta forma:

```
MultiPart multiPart = new MultiPart();  
multiPart.bodyPart("Mensagem com anexos",  
                    MediaType.TEXT_PLAIN_TYPE);
```

```
ClientBuilder  
    .newClient()  
    .register(MultiPartFeature.class)  
    .target("http://localhost:8080/cervejaria/services/email")  
    .request()  
    .header("To", "alesaudate@gmail.com")  
    .header("Cc", "fake@fake.com")  
    .header("Bcc", "fake2@fake.com")  
    .header("Subject", "Olá, mundo!")  
    .post(Entity.entity(multiPart, multiPart.getMediaType()));
```

Existe também uma facilidade no componente do Jersey que nos ajuda a enviar arquivos, por meio da classe:

```
org.glassfish.jersey.media.multipart.file.FileDataBodyPart.
```

Esta recebe como parâmetro uma string contendo o nome do arquivo e um `java.io.File`, que a API utilizará para ler o conteúdo do arquivo e enviar para o servidor:

```
URI uriDoArquivo =  
    Cliente.class.getResource("/Erdinger Weissbier.jpg").toURI();  
File arquivo = new File(uriDoArquivo);  
multiPart.bodyPart("Mensagem com anexos",  
    MediaType.TEXT_PLAIN_TYPE)  
.bodyPart(new FileDataBodyPart("imagem", arquivo));
```

Isso vai provocar uma saída semelhante à anterior, porém com o conteúdo do arquivo escrito.

9.4 SERVIÇOS ASSÍNCRONOS

Muitas vezes, o processamento a ser realizado por nossos serviços requer mais tempo do que uma conexão HTTP "comum" pode suportar. Quando temos tal situação em mãos, uma boa pedida seria utilizar serviços assíncronos. Ou seja, serviços em que o cliente não fica preso esperando uma resposta, mas pode efetuar outros processamentos enquanto aguarda.

Existem duas abordagens para resolução do problema. A versão 2 do JAX-RS introduz uma técnica para isso que faz com que a conexão fique aberta, mas outra *thread* faça o processamento, liberando a thread que atende o cliente para atender outros. Isto é realizado por meio da interface `javax.ws.rs.container.AsyncResponse`, em conjunto com a anotação `javax.ws.rs.container.Suspended`.

Uma vez que a requisição é recebida, a instância de `AsyncResponse` é passada como parâmetro para a thread. Quando o processamento é finalizado, basta invocar o método `resume`, passando como parâmetro o resultado. Por exemplo, suponha que você queira recuperar um determinado e-mail de um servidor (caso

típico para processamento assíncrono, já que envolve uma requisição para um sistema externo). Poderíamos implementar o caso assim:

```
import java.util.concurrent.*;
//outros imports omitidos

@Path("/email")
public class EmailService {

    private static ExecutorService executorService;

    static {
        executorService = Executors.newFixedThreadPool(20);
    }

    @GET
    @Produces(MediaType.APPLICATION_XML)
    public void recuperarEmails(
        @Suspended final AsyncResponse asyncResponse) {

        executorService.execute(new Runnable() {

            public void run() {
                try {
                    Thread.sleep(20 * 1000); //20 segundos
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                Email email =
                    new Email().withAssunto("Email recebido");
                asyncResponse.resume(email);

            }
        });
    }
}
```

Ou seja, a execução da thread do JAX-RS será liberada rapidamente, ao passo que a nova thread a ser executada levará 20 segundos para ser executada. Do lado do cliente, incluímos uma chamada para o método `async`, que vai informar ao cliente que

esta chamada é assíncrona. Isto também vai fazer com que a invocação retorne imediatamente. Entretanto, em vez de retornar uma `javax.ws.rs.core.Response`, vai retornar uma `java.util.concurrent.Future<Response>`:

```
Future<Response> futureResponse = ClientBuilder
    .newClient()
    .target("http://localhost:8080/cervejaria/services/email")
    .request()
    .async()
    .get();

System.out.println("Requisição submetida. Aguardando
    resposta...");
//A chamada ao método get vai bloquear até que haja um
//resultado.
//Também é possível fornecer como parâmetro o tempo que vamos
//esperar
Response response = futureResponse.get();

System.out.println("Resposta:");
System.out.println(response.getStatus());
System.out.println(response.readEntity(Email.class));
```

A execução deste código vai fazer com que a primeira linha ("Requisição submetida..") seja impressa imediatamente. O restante vai ser impresso após 20 segundos:

```
Requisição submetida. Aguardando resposta...
Resposta:
200
Email [mensagem=, formatoMensagem=text/plain,
    assunto>Email recebido, destinatarios=[], comCopia=[],
    comCopiaOculto=[], anexos=[]]
```

Observe que este código leva em conta conexões estabelecidas com a mesma instância de serviço atendendo ao cliente. Caso a requisição leve realmente muito tempo (questão de vários minutos ou horas), é necessário pensar em cenários de falha da instância do serviço. Sendo assim, vamos implementar o cenário de forma mais próxima ao pensado inicialmente pelo protocolo HTTP: o serviço retorna o código 202 imediatamente, com os cabeçalhos `Location`

e `Expires`.

O cabeçalho `Location` vai indicar ao cliente onde a resposta pode ser buscada, e o `Expires` indica o tempo, que pode não ser apurado — neste caso, o cliente deve estar preparado para que a resposta não esteja pronta mesmo após este tempo.

Sendo assim, precisamos de alguma forma de armazenamento temporário para armazenar a resposta. Vou realizar a implementação com um `java.util.Map` em memória, mas em sistemas reais, a implementação ideal seria a utilização de sistemas de *grid* de dados, como o `Memcached`, `JBoss Infinispan`, `Oracle Coherence` ou `VMWare Gemfire`.

```
@Path("/emailInterop")
public class EmailServiceInterop {

    private static final ExecutorService executorService;
    private static final Map<String, Email> emails;

    static {
        executorService = Executors.newFixedThreadPool(20);
        emails = new ConcurrentHashMap<>();
    }

    @GET
    public Response recuperarEmails() {
        //implementação
    }
}
```

Note que implementei o mapa utilizando uma *string* como chave. A ideia é usar uma *string* que possa referenciar esta requisição de maneira única e, assim, gerar uma URL também única para a qual o cliente pode solicitar o resultado desse processamento. Para gerar este ID, vamos utilizar UUID (*Universally Unique Identifier*), ou seja, uma *string* que, garantidamente, é única. Para gerá-la, usamos a classe `java.util.UUID`:

```
final String emailId = UUID.randomUUID().toString();
```

```

executorService.execute(new Runnable() {

    public void run() {
        try {
            Thread.sleep(20 * 1000); // 20 segundos
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        Email email = new Email().withAssunto("Email recebido");
        emails.put(emailId, email);
    }
});
```

Agora, é necessário gerar a resposta para o cliente. Para isso, precisamos criar um tempo de resposta, que vamos aumentar em um segundo para que o tempo seja suficiente de contar o tempo do

`Thread.sleep`, a geração do e-mail e mais o tempo de transferência das informações para o cliente. Também é necessário gerar o link onde o cliente pode recuperar a resposta da requisição:

```

Calendar calendar = Calendar.getInstance();

//Tempo do Thread.sleep + tempo para gerar o email
calendar.add(Calendar.SECOND, 21);

URI uri = UriBuilder
    .fromPath("/cervejaria/services")
    .path(EmailServiceInterop.class)
    .path("/{id}")
    .build(emailId);

Link link = Link.fromUri(uri).build();

return Response
    .accepted()
    .header("Location", link.getUri())
    .header("Expires", calendar.getTime())
    .build();
```

Pronto, isso vai gerar a resposta com status 202 para o cliente. O último passo é gerar o método onde o cliente pode ir para conferir o resultado:

```

@Path("/{id}")
@GET
@Produces({ APPLICATION_XML, APPLICATION_JSON })
public Response recuperaEmail(@PathParam("id") String id) {

    Email email = emails.get(id);
    if (email == null) {
        return Response.noContent().build();
    } else {
        emails.remove(id);
        return Response.ok(email).build();
    }
}

```

O cliente para este método fica assim:

```

Response response = ClientBuilder
    .newClient()
    .target("http://localhost:8080/cervejaria/services
            /emailInterop")
    .request()
    .get();

System.out.println("Status da resposta:");
System.out.println(response.getStatus());

String expires = response.getHeaderString("Expires");
System.out.println("Retornar a requisição em " + expires);
System.out.println("link recuperado: " +
    response.getHeaderString("Location"));

Date date = new DateProvider().fromString(expires);

long now = System.currentTimeMillis();
long waitUntil = date.getTime() - now;

System.out.println("Esperar " + (waitUntil )
    + " millissegundos");
Thread.sleep(waitUntil);

System.out.println("Ressubmetendo a requisição...");
response = ClientBuilder
    .newClient()
    .target(response.getHeaderString("Location"))
    .register(JettisonFeature.class)
    .request()
    .get();

```

```
System.out.println("Lido:");
System.out.println(response.readEntity(Email.class));
```

O que vai produzir a seguinte saída:

Status da resposta:

202

Retornar a requisição em Sun, 16 Mar 2014 23:41:07 GMT
link recuperado: http://localhost:8080/cervejaria/services

/emailInterop/8a067c31-fc3e-44e2-a473-7ebad7e9df02

Esperar 20377 millissegundos

Ressubmetendo a requisição...

Lido:

Email [mensagem=, formatoMensagem=text/plain,
assunto=Email recebido, destinatarios=[], comCopia=[],
comCopiaOculto=[], anexos=]

9.5 ATUALIZAÇÕES CONCORRENTES

Muitas vezes, nosso serviço é acessado concorrentemente por vários clientes. Quando isso acontece, precisamos ter um cuidado especial para fornecer aos clientes meios de não modificar por acidente uma informação. Por exemplo, suponha o seguinte cenário:

- Um cliente A solicitou uma informação X;
- Um cliente B solicitou a mesma informação X;
- O cliente A faz uma solicitação de atualização para a informação X, tornando-se assim X';
- O cliente B faz uma solicitação de atualização da informação X, mas o que o sistema possui é X' – assim sendo, o cliente B está tentando realizar uma atualização com base em uma informação imprecisa.

Para resolver o problema, o protocolo HTTP dispõe de uma funcionalidade chamada *Entity Tag* (também abreviada como *ETag*). Uma *ETag* é basicamente um *hash* do recurso em sua forma atual, incluso na resposta de uma solicitação no cabeçalho *ETag*.

De posse deste valor, o cliente pode checar se o estado do servidor é o esperado a partir dos cabeçalhos `If-Match` e `If-None-Match`, que vão fazer a checagem, respectivamente, se o recurso presente no servidor ainda tem o mesmo *hash* e se nenhum recurso presente no servidor atende a este *hash*. No mesmo exemplo fornecido anteriormente, a conversação ficaria assim:

- Um cliente A solicita uma informação X. Esta informação vem acompanhada do cabeçalho `ETag`, que tem o valor `ABC`;
- Um cliente B solicita a mesma informação, e o cabeçalho `ETag` vem com o mesmo valor;
- O cliente A faz uma solicitação de atualização, com o cabeçalho `If-Match` presente e valor `ABC`;
- Como o conteúdo presente no servidor ainda tem o mesmo *hash*, o cliente A recebe o status 200 ou 204 como resposta;
- O cliente B faz a solicitação de atualização, com o cabeçalho `If-Match` com o valor `ABC`;
- Como o cliente A conseguiu ter sucesso em sua atualização, o *hash* de X' é diferente de X;
- O servidor retorna para o cliente B, portanto, o código de status `409 Conflict`, indicando que este deve solicitar a informação novamente.

Assim sendo, a primeira coisa que temos de implementar é uma calculadora de *hashes* para nossas entidades. Vou utilizar as bibliotecas `commons-lang` e `commons-codec`, da Apache, para ajudar nesse processo. Preciso incluir o seguinte no meu arquivo `pom.xml`:

```
<dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>2.6</version>
</dependency>
```

```
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.9</version>
</dependency>
```

O próximo passo é implementar uma classe utilitária que vai calcular esse *hash* em cima dos objetos que eu fornecer. Para isso, vou criar uma classe Hash :

```
import java.nio.ByteBuffer;

import org.apache.commons.codec.binary.Hex;
import org.apache.commons.lang.builder.HashCodeBuilder;

public class Hash {

    public static String hash(Object entity) {
        int hashCode = HashCodeBuilder.reflectionHashCode(entity);

        //Converte o int em array de bytes
        byte[] array = ByteBuffer.allocate(4).putInt(hashCode)
            .array();

        //Converte o array em hexadecimal
        String hash = Hex.encodeHexString(array);
        return hash;
    }
}
```

O próximo passo é incluir o *hash* na resposta para o cliente. Para isso, basta invocar o método tag da classe Response , passando como parâmetro o valor calculado do *hash* :

```
import static br.com.geladaonline.util.Hash.hash;
//outros imports omitidos

public class CervejaService {

    @GET
    @Path("{nome}")

    public Response encontreCerveja(
        @PathParam("nome") String nomeDaCerveja) {
```

```

        Cerveja cerveja =
            estoque.recuperarCervejaPeloNome(nomeDaCerveja);
        if (cerveja != null) {
            return Response.ok(cerveja).tag(hash(cerveja)).build();
        }

        throw new WebApplicationException(Status.NOT_FOUND);

    }
}

```

Agora, precisamos de um método que vai fazer o teste em relação a atualizações sobre o recurso Cerveja :

```

private void testaETag(String eTag, String nomeCerveja) {
    Response responseEncontraCerveja =
        encontreCerveja(nomeCerveja);

    if (StringUtils.isNotEmpty(eTag)
        && !responseEncontraCerveja.getEntityTag()
            .getValue().equals(eTag)) {

        throw new WebApplicationException(Status.CONFLICT);

    }
}

```

Finalmente, modificamos nossos métodos de atualização para termos acesso ao cabeçalho `If-Match`. Note que este valor pode ou não ser fornecido (daí a necessidade de se testar se ele tem realmente um valor no método `testaETag`):

```

@PUT
@Path("{nome}")
public void atualizarCerveja(@PathParam("nome") String nome,
    @HeaderParam("If-Match") String eTag, Cerveja cerveja) {

    testaETag(eTag, nome);
    cerveja.setNome(nome);
    estoque.atualizarCerveja(cerveja);
}

```

9.6 CACHEAMENTO DE RESULTADOS

Uma outra *feature* muito útil é o cacheamento de resultados, especialmente se nossos serviços estiverem sendo acessados por meio de dispositivos móveis. Este cacheamento, assim como o uso de *ETags*, é baseado em cabeçalhos padronizados do protocolo HTTP e, portanto, interoperável. A técnica utilizada é a seguinte:

- O cliente A solicita a informação X;
- A resposta desta solicitação vem acompanhada de um cabeçalho `Date`, contendo a data em que o conteúdo foi gerado para este cliente;
- O cliente A armazena o resultado desta requisição, junto com a data em que foi obtida;
- Em uma requisição subsequente, o cliente A faz a mesma requisição, mas passando o cabeçalho `If-Modified-Since`, com o valor igual à data em que obteve os dados na primeira requisição;
- Se os dados não tiverem sido alterados por outro cliente, o cliente recebe o status `304 Not Modified`;
- Caso os dados tenham sido alterados, o cliente A recebe como resultado os mesmos dados, com o status `200 OK`.

Para implementar esta técnica, no entanto, vou levar em conta uma abordagem diferente. Em vez de implementar o cacheamento diretamente no código, criarei um interceptador para o recurso. Este interceptador vai ter as seguintes características:

- Caso o método utilizado pelo cliente seja GET, checa-se o cabeçalho `If-Modified-Since` está presente;
- Se estiver presente, mas o *cache* não reconhecer a referência (ou seja, não existe registro da informação no *cache*), delega-se a requisição para a implementação;
- Se o *cache* contiver o registro, checa-se a data de inclusão;

- Se o registro for posterior à data passada, então delega-se a requisição para a implementação;
- Se o registro for anterior, retorna-se o objeto cacheado para o cliente.

Vamos começar a implementação pelo *cache*. Note que a implementação de um *cache* é algo bastante sensível à memória e, assim como dito antes, você deve utilizar um *grid* de memória se a implementação estiver em *cluster*.

Se for uma máquina individual, recomendo utilizar um mapa apontando para instâncias referenciadas através da classe `java.lang.ref.SoftReference`. Esta possui uma implementação que faz com que, caso a JVM necessite de memória, as instâncias referenciadas por meio dela sejam limpadas, oferecendo um grande apoio em caso de necessidade de construção de aplicações sensíveis à memória:

```
public class EntityCache {

    private Map<String,
        SoftReference<EntityDatePair>> objectCache;

    public EntityCache() {
        this.objectCache = new ConcurrentHashMap<>();
    }
}
```

Agora, precisamos de uma classe para armazenar tanto o objeto a ser cacheado quanto sua data:

```
private static class EntityDatePair {

    private Object entity;
    private Date date;

    public EntityDatePair(Object entity, Date date) {
        this.entity = entity;
        this.date = date;
    }
}
```

```

public Object getEntity() {
    return entity;
}

public Date getDate() {
    return date;
}
}

```

Assim, vamos criar o método para realizar a inclusão de uma entidade no *cache*. Note que a chave é justamente a URL fornecida pelo cliente:

```

public void put(String path, Object entity) {
    EntityDatePair pair = new EntityDatePair(entity, new Date());
    SoftReference<EntityDatePair> sr = new SoftReference<>(pair);
    this.objectCache.put(path, sr);
}

```

Finalmente, precisamos implementar o método de checagem. Ele vai receber como parâmetro a chave (ou seja, a URL fornecida pelo cliente) e a data passada no cabeçalho `If-Modified-Since`. A partir desses parâmetros, vai retornar um `boolean` dizendo se o recurso foi atualizado desde então (retornando `true`) ou não (retornando `false`). Este método precisa efetuar a seguinte sequência de passos:

1. Recuperar a `SoftReference` a partir da URL;
2. Checar se a referência não foi limpada (ou seja, o método `get` deve retornar algo diferente de `null`);
3. Se a referência for nula, remover a referência por completo do mapa;
4. Depois, comparar as datas fornecidas. Para isso, também é necessário remover os milissegundos de precisão da data, já que as datas utilizadas no protocolo HTTP só têm a precisão em nível de segundos (diferente das datas em Java, que vão até os milissegundos);
5. Retornar `true` se a data armazenada for posterior à fornecida e `false` em caso contrário.

```

private static final long INTERVALO_CEGO = 1000L;

public boolean isUpdated(String path, Date since) {

    SoftReference<EntityDatePair> sr = objectCache.get(path);
    if (sr != null) { //a referência não foi inserida no mapa

        EntityDatePair pair = sr.get();
        if (pair == null) { // se a referência foi limpa pelo GC
            objectCache.remove(path);
            return true;
        }

        long tempoArmazenado =
            pair.getDate().getTime() / INTERVALO_CEGO;
        long tempoFornecido = since.getTime() / INTERVALO_CEGO;

        //se a data armazenada é posterior à data passada como
        //parâmetro, significa que o objeto foi alterado
        return tempoArmazenado > tempoFornecido;
    }
    return true;
}

```

Agora, vamos criar o interceptador. Para interceptar a entrada do método (ou seja, o intervalo entre a requisição do cliente e a chegada à implementação do serviço), usamos a interface `javax.ws.rs.container.ContainerRequestFilter` ; para interceptar a saída, utilizamos a interface `javax.ws.rs.container.ContainerResponseFilter` :

```

public class CacheInterceptor implements
    ContainerRequestFilter, ContainerResponseFilter {

    private EntityCache entityCache;
    private org.glassfish.jersey.message.internal.DateProvider
        dateProvider;

    public CacheInterceptor() {
        this.entityCache = new EntityCache();
        this.dateProvider = new DateProvider();
    }

    //Invocado na requisição

```

```

@Override
public void filter(ContainerRequestContext requestContext)
    throws IOException {

}

//Invocado na resposta

@Override
public void filter(ContainerRequestContext requestContext,
    ContainerResponseContext responseContext)
    throws IOException {

}
}

```

Para tratar a requisição, a interface `javax.ws.rs.container.ContainerRequestContext` tem condições de fornecer todos os dados de que precisamos, ou seja, o método HTTP utilizado, o valor do cabeçalho `If-Modified-Since` (se presente) e o recurso solicitado. Para recuperar o método usado, utilizamos o método `getMethod` desta interface:

```

if (requestContext.getMethod().equals("GET")) {
}

```

Para recuperarmos o cabeçalho `If-Modified-Since`, usamos o método `getHeaderString`:

```

String unparsedDate = requestContext
    .getHeaderString("If-Modified-Since");

```

A partir daqui, precisamos converter a data em `java.util.Date`. Isto pode ser feito a partir do método `fromString` da classe `org.glassfish.jersey.message.internal.DateProvider`:

```

Date date = dateProvider.fromString(unparsedDate);

```

E para recuperarmos o recurso solicitado pelo cliente, utilizamos o método `getUriInfo().getPath()`:

```
String path = requestContext.getUriInfo().getPath();
```

De posse destes dados, invocamos o método `isUpdated` no *cache* que criamos. Caso o recurso não tenha sido modificado, criamos uma resposta com o status 304 e invocamos o método `abortWith`:

```
if (!entityCache.isUpdated(path, date)) {
    Response response =
        Response.status(Status.NOT_MODIFIED).build();
    requestContext.abortWith(response);
}
```

A implementação completa do método fica assim:

```
public void filter(ContainerRequestContext requestContext)
    throws IOException {

    if (requestContext.getMethod().equals("GET")) {
        String unparsedDate = requestContext
            .getHeaderString("If-Modified-Since");

        if (StringUtils.isNotEmpty(unparsedDate)) {

            Date date = dateProvider.fromString(unparsedDate);
            String path = requestContext.getUriInfo().getPath();

            if (!entityCache.isUpdated(path, date)) {

                Response response = Response
                    .status(Status.NOT_MODIFIED).build();
                requestContext.abortWith(response);
            }
        }
    }
}
```

Agora, precisamos implementar a interceptação da resposta. Fazer isto é substancialmente mais fácil; precisamos recuperar o resultado da invocação do serviço e também a URL do recurso. Depois, adicionamos ambos ao *cache*:

```
@Override
public void filter(ContainerRequestContext requestContext,
    ContainerResponseContext responseContext)
```

```

    throws IOException {

Object entity = responseContext.getEntity();
String path = requestContext.getUriInfo().getPath();

entityCache.put(path, entity);
}

```

O último passo é fazer com que este interceptador seja adicionado ao contexto. Para fazer isso, voltamos ao método `getSingletons` na classe `ApplicationJAXRS`:

```

@Override
public Set<Object> getSingletons() {

Set<Object> singletons = new HashSet<>();
singletons.add(new JettisonFeature());
singletons.add(new CacheInterceptor());
return singletons;
}

```

Se usarmos o código desta maneira, todas as requisições para todos os serviços serão interceptadas. Podemos controlar isto criando uma anotação, anotando sua definição com `javax.ws.rs.NameBinding` e, então, anotando tanto o interceptador quanto os serviços:

```

import java.lang.annotation.*;
import javax.ws.rs.NameBinding;

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@NameBinding
public @interface Cached {}

@Cached
public class CacheInterceptor {

    //restante da implementação
}

@Cached
public class CervejaService {
    //restante da implementação
}

```

Para realizar o teste, basta realizar duas requisições: uma sem o cabeçalho `If-Modified-Since` e a seguinte com. Para cada uma das respostas, vamos verificar o código de status retornado e a entidade retornada:

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

Response response = client
    .target("http://localhost:8080/cervejaria/services")
    .path("/cervejas")
    .request(MediaType.APPLICATION_XML)
    .get();

Date responseDate = new Date();
System.out.println("Código: " + response.getStatus());

Cervejas cervejas = response.readEntity(Cervejas.class);

System.out.println("Resultado: " + cervejas);

response = client
    .target("http://localhost:8080/cervejaria/services")
    .path("/cervejas")
    .request(MediaType.APPLICATION_XML)
    .header("If-Modified-Since", responseDate)
    .get();

System.out.println("Código: " + response.getStatus());

cervejas = response.readEntity(Cervejas.class);

System.out.println("Resultado: " + cervejas);
```

O que deve produzir um resultado semelhante ao seguinte:

```
Código: 200
Resultado: br.com.geladaonline.model.rest.Cervejas@44a4072d
Código: 304
Resultado: null
```

9.7 TESTES AUTOMATIZADOS DE SERVIÇOS REST

Uma parte importante do desenvolvimento de qualquer

software é o desenvolvimento de testes automatizados, que vão garantir que a sua API continua funcionando mesmo depois de sofrer alterações. O Jersey facilita esse processo fornecendo um framework de testes próprio, que vai cuidar de inicializar um servidor de testes e autoconfigurar um cliente JAX-RS que vai enxergá-lo.

Para adicionar as dependências deste framework de testes, basta incluir o seguinte no `pom.xml` :

```
<dependency>
    <groupId>
        org.glassfish.jersey.test-framework.providers
    </groupId>
    <artifactId>
        jersey-test-framework-provider-grizzly2
    </artifactId>
    <version>${jersey.version}</version>
    <scope>test</scope>
</dependency>
```

O framework de testes do Jersey é, na verdade, uma extensão do JUnit, um dos mais populares frameworks de testes para Java. Quando realizamos a inclusão deste framework, portanto, automaticamente estamos incluindo o JUnit.

Para criar um novo teste, basta criar uma classe que vai realizar a execução deste teste e estender a classe `org.glassfish.jersey.test.JerseyTest` :

```
public class CervejaServiceIT extends JerseyTest {  
}
```

TESTES UTILIZANDO MAVEN

Lembre-se de que, seguindo as convenções do Maven, as classes de teste devem ser criadas na pasta `src/test/java`.

O próximo passo é sobrescrever o método `configure`, que vai retornar uma instância da classe `javax.ws.rs.core.Application`. Vamos aproveitar que a classe `ApplicationJAXRS` estende essa classe e retornar uma instância dela:

```
@Override  
protected Application configure() {  
    return new ApplicationJAXRS();  
}
```

Finalmente, criamos o teste. Este teste segue as regras do JUnit, ou seja, precisamos criar métodos com retorno `void` que estejam anotados com `org.junit.Test`:

```
@Test  
public void testeCenarioFeliz() throws Exception {  
    // implementação do teste  
}
```

Agora, vamos à implementação do teste propriamente dito. Note que o processo é bastante similar à criação dos clientes que fizemos até agora; mas não precisamos configurar o endereço nem o contexto ou qualquer coisa do tipo, basta apontar diretamente o teste para a URL de testes:

```
Cervejas cervejas = target("/cervejas")  
.register(JettisonFeature.class)  
.request().get(Cervejas.class);
```

Finalmente, utilizamos os métodos do JUnit para verificar se os resultados retornados são os esperados. Para isso, fizemos uso dos métodos (estáticos) presentes na classe `org.junit.Assert`. Para facilitar o processo, usamos uma importação estática destes métodos:

```
import static org.junit.Assert.*;
```

Os métodos desta classe vão lançar erros automaticamente caso o cenário esteja diferente do esperado. Por exemplo, para

garantirmos que a instância de `Cervejas` que obtivemos nessa invocação não é nula, utilizamos o método `assertNotNull`:

```
assertNotNull(cervejas);
```

OBSERVAÇÃO SOBRE A CLASSE CERVEJAS

Observe que a classe `Cervejas` que cito aqui é uma cópia da classe que criamos para atuar do lado do cliente, conforme apresentado no capítulo *REST, client-side*.

A partir daqui, é tudo um *script* para checar se o fluxo de obtenção das cervejas é o esperado, ou seja, se a primeira solicitação retorna dois links, apontando para a `Erdinger` e para a `Stella Artois`, e se os links trazem os dados corretos.

Para checar se os dois links existem, codificamos o seguinte:

```
Cervejas cervejas = target("/cervejas")
    .register(JettisonFeature.class)
    .request().get(Cervejas.class);
assertNotNull(cervejas);
assertNotNull(cervejas.getLinks());
assertFalse(cervejas.getLinks().isEmpty());
assertEquals(2, cervejas.getLinks().size());
```

Para checar se os links estão corretos:

```
Link link1 = cervejas.getLinks().get(0);
Link link2 = cervejas.getLinks().get(1);

String caminhoCompleto =
    "http://localhost:8080/cervejaria/services/";

assertEquals(
    caminhoCompleto + "cervejas/Erdinger Weissbier",
    java.net.URLDecoder.decode(link1.getUri().toASCIIString(),
        "UTF-8"));

assertEquals(caminhoCompleto + "cervejas/Stella Artois",
```

```

java.net.URLDecoder.decode(link2.getUri().toASCIIString(),
                           "UTF-8"));

assertEquals("cerveja", link1.getRel());
assertEquals("cerveja", link2.getRel());

assertEquals("Erdinger Weissbier", link1.getTitle());
assertEquals("Stella Artois", link2.getTitle());

```

Para checar se o primeiro link efetivamente traz os dados esperados:

```

Cerveja erdinger = target("cervejas/Erdinger Weissbier")
    .register(JettisonFeature.class)
    .request(MediaType.APPLICATION_XML).get(Cerveja.class);

assertNotNull(erdinger);
assertEquals("Erdinger Weissbier", erdinger.getNome());
assertEquals("Erdinger Weissbräu", erdinger.getCervejaria());
assertEquals("Cerveja de trigo alemã", erdinger.getDescricao());
assertEquals(Cerveja.Tipo.WEIZEN, erdinger.getTipo());

```

Para checar se o segundo link também traz os dados esperados:

```

Cerveja stella = target("cervejas/Stella Artois")
    .register(JettisonFeature.class)
    .request(MediaType.APPLICATION_XML).get(Cerveja.class);

assertNotNull(stella);
assertEquals("Stella Artois", stella.getNome());
assertEquals("Artois", stella.getCervejaria());
assertEquals("A cerveja belga mais francesa do mundo :)",
            stella.getDescricao());
assertEquals(Cerveja.Tipo.LAGER, stella.getTipo());

```

De forma semelhante, implementamos outros testes sobre o serviço. Por exemplo, para testarmos a inclusão da cerveja, temos de checar se o código de status retornado é `201 Created`, se o cabeçalho `Location` está presente e se a URL presente neste cabeçalho realmente aponta para a cerveja recém-criada:

```

@Test
public void testInsertCerveja() {

    Cerveja skol = new Cerveja("Skol",
        "Cerveja dinamarquesa abrasileirada", "Ambev",

```

```

        Cerveja.Tipo.PILSEN);

Response response = target("/cervejas").request()
    .post(Entity.xml(skol));

assertNotNull(response);
assertEquals(201, response.getStatus());
assertNotNull(response.getLocation());

assertTrue(response.getLocation().toASCIIString()
    .endsWith("/cervejas/Skol"));

skol = target("/cervejas/Skol")
    .request(MediaType.APPLICATION_XML)
    .get(Cerveja.class);

assertEquals("Skol", skol.getNome());
assertEquals("Cerveja dinamarquesa abrasileirada",
    skol.getDescricao());
assertEquals(Cerveja.Tipo.PILSEN, skol.getTipo());
assertEquals("Ambev", skol.getCervejaria());
}

}

```

9.8 CRIANDO PARSERS CUSTOMIZADOS DE DADOS

Um dos últimos pontos a serem levados em consideração no desenvolvimento de serviços em REST é ter em mente que, muitas vezes, é preciso trabalhar com formatos de dados que não são analisados nativamente pela engine do JAX-RS. Por exemplo, quando retornamos um XML, a implementação do nosso código retorna objetos do nosso próprio modelo e a engine realiza a transformação destes dados em XML. Isto porque os transformadores são nativos da engine.

Mas o que aconteceria se precisássemos gerar um relatório em PDF, por exemplo? Este transformador não existe.

Assim sendo, criamos nossos próprios transformadores, que vão passar a ter este trabalho em vez de termos de codificar isto

diretamente no serviço, prejudicando a clareza da implementação. Vamos ver nesta seção como modificar o nosso serviço de cervejas para tratar mais adequadamente a visualização de imagens de cervejas.

Para ajudar neste tratamento, vamos incluir uma biblioteca da Apache especializada em I/O, a commons IO :

```
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.4</version>
</dependency>
```

Agora, vamos à implementação. Para realizar a transformação de uma entidade que sai do serviço, temos de implementar a interface `javax.ws.rs.ext.MessageBodyWriter`. Ela define três métodos: `isWriteable`, `getSize` e `writeTo`.

O método `isWriteable` checa se o transformador é apropriado para esta classe e MIME Type; o `getSize` costumava ser usado para checar o tamanho dos dados depois da transformação, mas foi depreciado; e, finalmente, o método `writeTo` realiza a operação de transformação de fato. A assinatura dos três métodos é a seguinte:

```
import java.io.*;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.*;
import javax.ws.rs.core.*;
import javax.ws.rs.ext.*;

import br.com.geladaonline.model.Cerveja;

@Provider
public class ImageProducer
    implements MessageBodyWriter<Cerveja> {

    @Override
    public boolean isWriteable(Class<?> type, Type genericType,
```

```

        Annotation[] annotations, MediaType mediaType) {
    // implementação
}

@Override
public long getSize(Cerveja t, Class<?> type,
    Type genericType, Annotation[] annotations,
    MediaType mediaType) {
    return -1L;
}

@Override
public void writeTo(Cerveja t, Class<?> type,
    Type genericType, Annotation[] annotations,
    MediaType mediaType, MultivaluedMap<String,
    Object> httpHeaders, OutputStream entityStream)
    throws IOException, WebApplicationException {
    //implementação
}
}
}

```

Vamos começar a implementação pelo método `isWriteable`. Antes, vamos analisar os parâmetros: o parâmetro `type` é a classe da entidade a ser transformada, o `genericType` também, mas obtido via `reflections`, o `annotations` são anotações presentes na classe da entidade, e o `mediaType` é o *media type* ajustado para ser retornado via HTTP. Assim sendo, precisamos testar se o parâmetro `type` é da classe `Cerveja`, e se o *media type* começa com `image`:

```

@Override
public boolean isWriteable(Class<?> type, Type genericType,
    Annotation[] annotations, MediaType mediaType) {

    boolean response = type.equals(Cerveja.class);
    response &= mediaType.getType().equals("image");
    return response;
}

```

A implementação do método `getSize`, como disse antes, foi depreciada pelo JAX-RS 2. Simplesmente retorne `-1`. Quanto ao método `writeTo`, seus parâmetros são: `t` é a instância retornada pelo serviço; `type`, `genericType`, `annotations` e `mediaType`

são iguais ao método `isWriteable` ; `httpHeaders` é um mapa contendo os HTTP Headers que foram incluídos pela implementação do serviço (e também é possível incluir novos).

Finalmente, `entityStream` é uma *stream* utilizada para escrever os dados para o cliente, e deve ser usada pelo transformador. Este método deverá recuperar o nome da cerveja a partir da instância e recuperar a imagem correspondente do *filesystem*. Se a imagem não existir, lance uma exceção com o código `404 Not Found`. Se for encontrada, escreva os dados da *stream* para o cliente e fecha a *stream* com os dados do arquivo encontrado.

Não é necessário fechar a `entityStream`, a engine do JAX-RS fará isso automaticamente:

```
@Override  
public void writeTo(Cerveja t, Class<?> type, Type genericType,  
    Annotation[] annotations, MediaType mediaType,  
    MultivaluedMap<String, Object> httpHeaders,  
    OutputStream entityStream) throws IOException,  
    WebApplicationException {  
  
    InputStream stream =  
        ImageProducer.class.getResourceAsStream("/" + t.getNome()  
            + ".jpg");  
  
    if (stream == null) {  
        throw new WebApplicationException(Status.NOT_FOUND);  
    }  
  
    IOUtils.copy(stream, entityStream);  
    IOUtils.closeQuietly(stream);  
}
```

Para que a engine do JAX-RS detecte esta classe, podemos habilitar autodetecção por meio da anotação `javax.ws.rs.ext.Provider`:

```
@Provider  
public class  
    ImageProducer implements MessageBodyWriter<Cerveja> {  
    //...
```

```
}
```

Agora, podemos modificar a resposta do método `recuperaImagem` na classe `CervejaService`:

```
@GET  
@Path("{nome}")  
@Produces("image/*")  
public Response  
    recuperaImagem(@PathParam("nome") String nomeDaCerveja)  
throws IOException {  
  
    Cerveja cerveja = new Cerveja();  
    cerveja.setNome(nomeDaCerveja);  
  
    return Response.ok(cerveja).type("image/jpg").build();  
}
```

A implementação está pronta! Para testar, basta realizar a solicitação da imagem pelo `curl`, por exemplo.

Agora, vamos trabalhar com o processo inverso, ou seja, realizar a transformação dos dados fornecidos pelo cliente para uma classe nossa. Para isso, antes vamos criar uma classe que armazene estes dados:

```
public class Imagem {  
  
    public static final Map<String, String> EXTENSOES;  
  
    static {  
        EXTENSOES = new HashMap<>();  
        EXTENSOES.put("image/jpg", ".jpg");  
    }  
  
    private byte[] dados;  
  
    private String nome;  
  
    private MediaType mediaType;  
  
    public Imagem(byte[] dados, String nome,  
        MediaType mediaType) {  
        this.dados = dados;  
        this.nome = nome;
```

```

        this.mediaType = mediaType;
    }
}

```

Para criar o transformador dos dados do cliente, temos de implementar a interface `javax.ws.rs.ext.MessageBodyReader`. Ela define dois métodos: `isReadable` e `readFrom`. As assinaturas são muito semelhantes aos métodos `isWriteable` e `writeTo`, respectivamente:

```

public class ImageConsumer implements MessageBodyReader<Imagem>{

    @Override
    public boolean isReadable(Class<?> type, Type genericType,
                             Annotation[] annotations, MediaType mediaType) {

        //implementação
    }

    @Override
    public Imagem readFrom(Class<Imagem> type, Type genericType,
                           Annotation[] annotations, MediaType mediaType,
                           MultivaluedMap<String, String> httpHeaders,
                           InputStream entityStream)
        throws IOException, WebApplicationException {

        //implementação
    }
}

```

O método `isReadable` deve checar se a classe para a qual vai ser feita a escrita é `Imagem`, e se o *media type* é do tipo macro `image`:

```

@Override
public boolean isReadable(Class<?> type, Type genericType,
                         Annotation[] annotations, MediaType mediaType) {

    return type.equals(Imagem.class)
        && mediaType.getType().equals("image");
}

```

Agora, vamos realizar a leitura dos dados. Precisamos checar se a imagem está presente no mapa de extensões presente na classe

`Imagen` (se não estiver, retornamos o código `415 Unsupported Media Type`). Também podemos optar por receber o nome da imagem a partir de um cabeçalho, `nome`. Então, faremos a leitura dos dados fornecidos pelo cliente e faremos a inserção destes dados em uma instância da classe `Imagen`:

```
@Override  
public Imagem readFrom(Class<Imagen> type, Type genericType,  
    Annotation[] annotations, MediaType mediaType,  
    MultivaluedMap<String, String> httpHeaders,  
    InputStream entityStream)  
throws IOException, WebApplicationException {  
  
    if (!Imagen.EXTENSOES.containsKey(mediaType.toString())) {  
        throw new WebApplicationException(Status  
            .UNSUPPORTED_MEDIA_TYPE);  
    }  
  
    String nome = httpHeaders.getFirst("nome");  
  
    byte[] dados = IOUtils.toByteArray(entityStream);  
  
    Imagem imagem = new Imagem(dados, nome, mediaType);  
  
    return imagem;  
}
```

Vamos criar o método `salvar` na classe `Imagen`, para delegarmos a esta a capacidade de salvar os dados contidos:

```
public void salvar(String caminho) throws IOException {  
  
    FileOutputStream fos = new FileOutputStream(caminho  
        + java.io.File.separator + nome  
        + EXTENSOES.get(mediaType.toString()));  
    IOUtils.write(dados, fos);  
    IOUtils.closeQuietly(fos);  
}
```

Finalmente, alteramos a implementação do método `criaImagen` na classe `CervejaService`, para realizar a gravação desta imagem em disco:

```
@POST  
@Consumes("image/*")
```

```
public Response criaImagem(Imagen imagem) throws IOException,  
    InterruptedException {  
    imagem.salvar(System.getProperty("user.home"));  
    return Response.ok().build();  
}
```

Finalmente, está pronto. Nossos transformadores estão prontos para uso. Lembre-se de usar este recurso de forma sadia, ou seja, de maneira que seja o mais reutilizável possível.

9.9 CONCLUSÃO

Neste capítulo, você aprendeu diversas técnicas utilizadas em APIs REST que são utilizadas em cenários profissionais. Você viu:

- Técnicas para modelar cenários considerados complexos;
- Como utilizar serviços assíncronos;
- Como tratar serviços que têm muitos clientes alterando os mesmos dados;
- Como o cliente pode realizar cacheamento de resultados;
- Como criar testes automatizados de serviços;
- Como criar transformadores customizados de dados.

Com estas técnicas, você tem acesso a praticamente todas as informações de que precisa para construir APIs excelentes. Lembre-se de que este é um campo que permanece em estudos e, assim sendo, este livro não abordou todas as informações que existem a respeito disso, apenas as mais importantes.

CAPÍTULO 10

PERGUNTAS FREQUENTES SOBRE REST

Gostaria de encerrar este livro com alguns questionamentos que recebo com certa frequência. Como sou instrutor de SOA, em geral meus alunos costumam ter várias perguntas a respeito de REST. Essas perguntas sempre são pertinentes a cenários reais, presentes na realidade brasileira.

Assim sendo, elenco aqui alguns desses questionamentos com minhas respostas. Note que, se desejar, você também pode me mandar suas perguntas a respeito de REST no fórum oficial do livro: <https://groups.google.com/forum/#!forum/rest-construa-apis-inteligentes-de-maneira-simples>.

10.1 É POSSÍVEL USAR REST EM LARGA ESCALA?

Sim, perfeitamente possível. Existem APIs REST utilizadas em serviços amplamente usados, como no Netflix, Facebook, Twitter e outros.

Também é possível usar dentro de empresas; no entanto, assim como qualquer outra tecnologia deste tipo, requer graus de disciplina e maturidade para organizar a utilização das APIs. Uma boa pedida seria a organização de um comitê regulador da exposição destes serviços (conhecido tradicionalmente na

bibliografia de SOA como COE — *Center of Excellence*).

Alguns dos desafios que podem ser encontrados pela adoção de REST dentro de empresas são:

- Exposição de serviços duplicados;
- Problemas para lidar com evolução de serviços (incompatibilidade entre clientes e servidores);
- Documentação pobre / desatualizada / inexistente;
- Falta de domínio sobre as técnicas de REST (especialmente HATEOAS).

Para ter uma solução satisfatória para os três primeiros, a existência de um comitê regulador pode ajudar. O último tópico só pode ser resolvido por meio da disciplina dos próprios programadores. Espero que este livro possa auxiliar nesse sentido.

10.2 COMO GERENCIAR MÚLTIPLAS VERSÕES DE SERVIÇOS?

Existem várias técnicas para ajudar nisso, mas a mais aceita (e disseminada) é a inclusão da versão da API na própria URL dos serviços (algo como `/v1/usuarios`). Isto por parte do cliente. Do lado da implementação, este ponto é mais complexo.

Uma abordagem seria fazer diferenciação na própria implementação. É possível criar vários *servlets* distintos para atender às requisições dos clientes, mas o principal problema seria a garantia de separação da implementação dos serviços. Este caso seria mais recomendado em casos onde os incrementos de versões seriam pequenos, tal como adição de *query parameters* que não afetem versões anteriores.

Em incrementos de versão médios, poderia ser recomendado manter as implementações em WARs (ou arquivos de aplicações

equivalentes para outras linguagens que não Java) distintas, e implantadas separadamente. Para que o cliente consiga realizar diferenciação, inclua regra de tratamento em um *load balancer*, tal como Apache ou Nginx.

O problema com esta abordagem seria a infraestrutura compartilhada, ou seja, ainda assim não é possível garantir que uma versão não influencia o funcionamento de outra. Aqui, o *deploy* de uma mesma aplicação na mesma infraestrutura não necessariamente é um problema, pois as portas devem ser distintas (o que já é suficiente para diferenciar uma aplicação de outra). O *load balancer* deve ser responsável por resolver estas questões de roteamento.

Em incrementos de versão grandes, portanto, o ideal seria manter uma infraestrutura separada para cada versão da aplicação. O problema desta abordagem é o custo de se manter todo o equipamento separado.

Em suma, não há uma solução simples para isso. Incrementos de versão são inevitáveis. Com a experiência, você deve ganhar conhecimento suficiente para que estes incrementos tenham o mínimo impacto possível em incrementos de versão, de forma que seja possível fazer esta migração entre versões de forma sustentável.

Além da questão da implementação, também recomendo fortemente que você avise o maior número de pessoas possível a respeito de quaisquer modificações nos seus serviços. Aqui, o uso de AtomPub (<http://atompub.org/>) é uma boa pedida.

10.3 QUANTOS E QUAIS LINKS HATEOAS DEVO INSERIR NOS MEUS RECURSOS?

Uma técnica simples para facilitar a visualização dos links entre

os recursos é visualizar as ligações entre eles como grafos. Ou seja, se um usuário tem vários endereços, você deve visualizá-lo como um vértice do grafo e os endereços como outros vértices. Os links HATEOAS são equivalentes às arestas.

Da mesma forma, você deve visualizar o fluxo de ações que podem ser realizadas (modelagem baseada em BPMN definitivamente ajuda). As transições entre as ações também podem ser modeladas como links HATEOAS.

10.4 QUANDO EU DEVO PREFERIR UTILIZAR REST EM VEZ DE WS-*?

Antes de responder a esta pergunta, gostaria de reservar-me o direito de me declarar agnóstico, sem preferências. Gosto tanto de WS-* quanto de REST, pois enxergo vantagens e desvantagens em ambos.

Obviamente, REST atende melhor a cenários de serviços realmente voltados à Web. Em contraposição a WS-*, que atende melhor a cenários enterprise.

Sendo assim, obviamente REST possui grande vantagem em questões como:

- Performance. Como observado no livro, REST tira proveito de recursos do protocolo HTTP e possui cacheamento nativo. Além disso, pode utilizar recursos mais leves para transporte das informações, como JSON.
- Concorrência. Também foi observado no livro que REST tem suporte nativo a clientes concorrentes.
- Proteção de credenciais por meio de OAuth.

Ou seja, em cenários atuais, como clientes *mobile* acessando

servidores remotos, é praticamente obrigatório o uso de REST.

No entanto, algumas desvantagens de REST em relação a WS-* estão em questões como:

- Segurança avançada, com uso de *Single Sign-On*, por exemplo;
- Composições de serviços REST, pois existem poucas ferramentas especializadas em manutenção de transações entre serviços REST (apesar de existirem técnicas excelentes neste sentido). Mesmo assim, a modelagem de cenários deste tipo também é complexa;
- Serviços assíncronos (apesar de haver técnicas, como algumas que apresentei neste livro, ainda são rudimentares em relação ao que existe em WS-Addressing);

Note que estes cenários costumam aparecer com maior frequência apenas em cenários internos de empresas, demonstrando com precisão as diferenças entre WS-* e REST. Além disso, apesar dela possuir esta diferenciação, aplicar estas técnicas costuma ser razoavelmente complexo, fazendo de REST sempre uma ótima solução em cenários que precisam de respostas rápidas às demandas, como em times que aplicam metodologias ágeis, por exemplo.

10.5 QUANDO EU DEVO PREFERIR UTILIZAR REST EM VEZ DE COMUNICAÇÃO NATIVA DA MINHA LINGUAGEM DE PROGRAMAÇÃO?

Sempre que houver necessidade de comunicação pelo lado de fora da sua aplicação, permita-se utilizar REST. Note que, apesar de

alguns protocolos nativos (como RMI/IIOP, usado por *Enterprise JavaBeans* — EJBs) possuírem performance superior, estes requerem liberações específicas de firewall e possuem seus próprios problemas. Como REST utiliza HTTP, é mais fácil de debugar e realizar esta liberação usando REST.

Quanto à questão de performance, vivemos em momentos em que hardware e equipamentos mais eficientes de rede são cada vez mais baratos. Assim, a principal justificativa poderia ser em caso de necessidade extrema de performance e inexistência da necessidade de se configurar este tipo de equipamento.

CONCLUSÃO

O REST que você viu ao longo deste livro significa *REpresentational State Transfer*. No entanto, a palavra *rest* também existe no inglês, e significa "descanso". Isso leva a pensar que Roy Fielding tinha senso de humor, e queria criar algo que remetesse tão bem à ideia de simplicidade que os usuários dessa técnica realmente pensassem que teriam momentos para relaxar em vez de ficar acertando pequenos detalhes em outros mecanismos.

Na prática, contudo, é um tema que está permanentemente se expandindo e requer estudos. Por isso, não deixe de se manter sempre atualizado. Um bom ponto de partida para investigação destes tópicos está no meu próprio GitHub: <https://github.com/alesaudate/rest>.

Lá, você pode baixar todo o código deste livro e também fazer modificações à vontade para satisfazer as suas necessidades. Se tiver dúvidas, você também pode entrar em contato comigo no fórum oficial deste livro: <http://forum.casadocodigo.com.br/>.

Outras grandes fontes de conhecimento são o InfoQ (<http://www.infoq.com/>), Stack Overflow (<http://pt.stackoverflow.com/>), SOA Cloud (<http://soacloud.com.br/>) e GUJ (<http://www.guj.com.br/>). Obviamente, também não se esqueça de conferir a bibliografia deste livro. Procure por autores como Roy Fielding, Leonard Richardson, Cesare Pautasso e Martin Fowler.

Um abraço!

Alexandre Saudate

CAPÍTULO 12

REFERÊNCIAS BIBLIOGRÁFICAS

BORENSTEIN, Nathaniel; FREED, Ned. *Multipurpose internetmail extensions (mime) part two: Media types.* 1996. Disponível em: <https://tools.ietf.org/html/rfc2046>.

CROCKFORD, Douglas. *The application/json media type for JavaScript object notation (JSON).* 2006. Disponível em: <https://tools.ietf.org/html/rfc4627>.

EVANS, Eric. *Domain-driven design: Tackling complexity in the heart of software.* Addison-Wesley, 2003.

FIELDING, Roy Thomas. *Architectural styles and the design of networkbased software architectures.* 2000. Disponível em: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

HADLEY, Marc. *Web application description language.* 2009. Disponível em: <https://www.w3.org/Submission/wadl/>.

HAMMER, Eran. *The oauth 1.0 guide.* 2011.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *Open systems interconnection – basic reference model: The basic model.* 1994.

JOHNSON, Ralph; VLASSIDES, John; GAMMA, Erich; HELM,

Richard. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1994.

KOHN, D.; MURATA, M.; LAURENT, S. *XML media types*. 2001.

MASINTER, Larry; BERNERS-LEE, Tim; FIELDING, Roy. *Uniformresource identifier (URI): Generic syntax*. 2005.

NIELSEN, Henrik Frystyk; BERNERS-LEE, Tim; FIELDING, Roy. *Hypertext transfer protocol – http/1.0*. 1996.

PEON, Roberto; BELSHE, Mike. *Spdy protocol*. 2012.

RUBY, Sam; RICHARDSON, Leonard. *RESTful web services*. O'Reilly Media, 2007.

SAUDATE, Alexandre. *Soa aplicado: integrando com web services e além*. Casa do Código, 2013.